1

Final Report to Office of Naval Research

N00014-89-J-1168

Methodological Foundations for Designing Intelligent Computer-Based Training

September 3, 1991

Principal Investigator: Alan Lesgold

The above-titled grant permitted considerable further work, beyond an earlier contract, on tools for developing intelligent computer-based training. As a result of the grant, we were in a strong position to develop two generations of intelligent training systems for use by the Defense Department. While direct development of those systems was funded by the Air Force, the experience and infrastructural development needed for the work to be possible came out of the ONR grant. From the two systems, Sherlock I and II, we are extracting a large number of tools that will be made publicly available (they will be deposited with NTIS, and we expect to arrange commercial distribution at costs designed to be low but to cover distribution and support). In a strong sense, therefore, the project continues, but it no longer needs additional funds to proceed.

The remainder of this report lists the publications that described work facilitated by this grant and presents a brief description of the tools now being completed. When those tools are finished, a report with further details will be made available.

Publications Related to the Grant

Glaser, R., Lesgold, A. M., & Gott, S. (1991). Implications of Cognitive Psychology for Measuring Job Performance. In A. K. Wigdor & B. F. Green, Jr. (Eds.), *Performance assessment for the workplace: technical issues* (pp. 1-26). Washington, DC: National Academy Press.

Lesgold, A., Bonar, J. G., & Ivill, J. (1989). Toward intelligent systems for testing. In L B. Resnick (Ed.), *Knowing, learning, and instruction: Essays in Honor of Robert Glaser.* Hillsdale, NJ: Erlbaum.

Lesgold, A., Lajoie, S., Logan, D., & Eggan, G. (1990). Applying cognitive task analysis and research methods to assessment. In N. Frederiksen, R. Glaser, A. M. Lesgold, & M. Shafto (Eds.), *Diagnostic monitoring of skill and knowledge acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Lesgold, A. M., Lajoie, S. P., Bunzo, M., & Eggan, G. (1991). SHERLOCK: A coached practice environment for an electronics troubleshooting job. In J. Larkin and R. Chabay, & C. Scheftic (Eds.), *Computer assisted instruction and intelligent tutoring systems: Shared issues and complementary approaches.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Lesgold, A. (1990). Tying development of intelligent tutors to research on theories of learning. In H. Mandl, E. De Corte, S. N. Bennett, & H. F. Friedrich (Eds.), *Learning and instruction: European research in an international context.* Vol. 3. Oxford: Pergamon.

Lajoie, S., & Lesgold, A. (1990). Apprenticeship Training in the Workplace: Computer Coached Practice Environment as a New Form of Apprenticeship. *Machine-Mediated Learning, 3,* 7-28.

Lesgold, A., & Lajoie, S. (1991). Complex problem solving in electronics. In R. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms.* Hillsdale, NJ: Erlbaum.

Lesgold, A., Chipman, S., Brown, J. S., & Soloway, E. (1990). Prospects for information science and technology focused on intelligent training systems concerns. *Annual Review of Computer Science.* Pp. 383-394. Palo Alto, CA: Annual Review Press.

Lesgold, A. (1990). Intelligent computer aids for practice of complex troubleshooting. *Proceedings of the 1990 FAA Symposium on Training Technology.* Oklahoma City: Federal Aviation Administration Academy.

Lesgold, A. (In press). Assessment of intelligent training systems: Sherlock as an example. In E. Baker and H. O'Neil, Jr. (Eds.), *Technology assessment: Estimating the future.* (Tentative title). Hillsdale, NJ: Erlbaum.

Lesgold, A., Eggan, G., Katz, S., & Rao, G. (in press). Possibilities for Assessment Using Computer-Based Apprenticeship Environments. To appear in W.

Regian and V. Shute (Eds.), *Cognitive approaches to automated instruction.* Hillsdale, NJ: Erlbaum.

Lajoie, S. P., & Lesgold, A. M. (Accepted; revision being completed). Dynamic assessment of proficiency for solving procedural knowledge tasks. *Educational Psychologist.*

Lesgold, A. (in press). Research methodology in the postinformatic age. *Behavior Research Methods, Instruments, and Computers.*

Katz, S., & Lesgold, A. (in press). The role of the tutor in computer-based collaborative learning situations. To appear in S. Lajoie & S. Derry (Eds.), *Computers as cognitive tools.* Hillsdale, NJ: Lawrence Erlbaum Associates.

In Preparation[1]

Lesgold, A., Katz, S., Greenberg, L., Hughes, E., & Eggan, G. Extensions of intelligent tutoring paradigms to support collaborative learning. (in preparation). For presentation at a NATO Advanced Study Workshop on Cognitive Science and Medical Education and subsequent proceedings volume. June, 1991. Lucca, Italy.

Lesgold, A., & Katz, S. (in preparation). Models of cognition and educational technologies: Implications for medical training. Opening address for NATO Advanced Study Workshop on Educational Technology and subsequent proceedings volume. July, 1991. Enschede, Netherlands.

Eggan, G., & Lesgold, A. (in preparation). Modelling requirements for intelligent training systems. For presentation at NATO Advanced Study Workshop on Educational Technology and subsequent proceedings volume. July, 1991. Enschede, Netherlands.

Lesgold, A. (in preparation). An object-based situational approach to task analysis. For presentation at a NATO Advanced Study Workshop on learning electricity and electronics with advanced technology. June, 1991. Marne-la-Vallee, France.

Katz, S., & Lesgold, A. (in preparation). Modeling the student in Sherlock II. Paper for IJCAI workshop on agent modeling, Sydney, Australia, August, 1991. Publication of workshop proceedings is planned.

Lesgold, A. (in preparation). *Helping people learn complex jobs: Cognitive science and situated learning* (tentative title). To be published by Cambridge University Press, New York.

[1] Copies of full drafts of all papers listed as "in preparation" are available from Alan Lesgold at the address shown in this curriculum vitae. These are for comments only and should not be cited without permission. Drafts of the book are not yet available.

Tools Developed in this Project

During the course of the work on Sherlock II, we developed a large number of special tools to facilitate our work. These were developed in several generations, and a further revision is currently in progress. In this report, we list the main categories of tools that were developed. Two appendices consist of documents still undergoing revision that describe the new, more-integrated approach now being pursued.

Directed Graph Object and Its Implications

Sherlock I was written in Xerox Lisp, using an object-oriented programming tool called Loops. When we began planning Sherlock II, it was already clear that the specialized hardware environments needed for the first generation of cognitive technology tools was beginning to pose serious barriers to the diffusion of the ideas in those tools. We decided that we would do future development work on standard workstation platforms with broad presence in the commercial world. The Air Force Desktop contract series ended up determining our hardware choice, which was to use DOS machines with significant extended memory (memory was clearly becoming a cheap commodity and is not a barrier to the diffusion of our ideas). The decision to use Smalltalk for our work was based upon a desire to preserve the power of the object-based approach that was permitted by Loops, but in a more standard environment.

Comparing Smalltalk to Lisp, the most obvious difference is that the general list processing capabilities of Lisp are absent in Smalltalk. So, we decided to add them. However, in doing so, Maria Gordin, a software designer in our staff, pointed out that lists are really themselves specializations of a more general category, directed graphs. Gordin proceeded to implement a set of directed graph manipulation methods that could be part of a directed graph object type. With these methods, we can do everything Lisp does, and more besides, since we can handle easily structures that include cycles, such as circuit representations. With Lisp list structures, representation of a circuit is possible, and not much more complex, but with the directed graph data type as the core primitive, we focus optimization efforts at a more productive level. For example, searching a directed graph structure proceeds as quickly in one direction as in another, while in Lisp backward searches are significantly more complex. Indeed, we suspect that if Lisp were being redesigned today, with memory and processor power cheap, we would likely see directed graphs as its primitive type, too.

There are a number of specializations of the directed graph type that we have developed. These include circuit simulations, student models, graphic forms, version control data structures, flowcharts, etc.

Circuit simulations are an obvious case. A circuit, after all, can be modeled as a recursive combination of entities that each have certain properties. Each entity in a circuit has two or more information ports connecting it to other entities. When a change occurs in the information present at one of its ports, the simulation for an entity must know how to determine the changes that will occur at other ports (and possibly their temporal dynamics). Such a representation is recursive, since each entity can itself be modeled as a network of smaller entities.

At first glance, it might appear that the relevant graph structures for a circuit should be undirected. However, in practice, causal explanations of circuit function make use, at least, of context-bound directionality. De Kleer and Brown's concept of mythical causality requires that the implications of events be propagated as part of an explanation. For many purposes, the contexts in which an explanation is required imposes strong directionality on the circuit. For example, we model test stations in Sherlock II. A test station is a large switch that applies patterned electrical energy to a device that is being tested and then transfers a signal from some part of that device to a measurement instrument. Even though a deep explanation of the resulting circuit created by a test configuration may not benefit from any directionality of circuit connections, an account of its function does imply a strong directionality from power supplies to device being tested to measurement instrument.

Student model and browser. Because of the centrality of student modeling to cognitive technology training applications and research, we also have developed a student modeling scheme that is a specialization of the directed graph object and also uses "browser" tools built into Sherlock. The student modeling scheme we developed is described in Appendix A, which is a paper presented at a 1991 IJCAI workshop. As indicated in the Appendix, we have developed a rich browsing capability that allows easy exploration of the complex of variables, at multiple levels of abstraction, that represent a student's performance. In addition, for any variable, one can review where its score comes from (local variables are determined by updating rules which directly relate cognitive outcomes that are observable to the variables' values; more abstracted variables are aggregations of local variables). As can be seen from the illustrations in Appendix A, the lattice structure of the student model is illustrated graphically. In addition, a "stack" is maintained to help in navigating through the student model graph, thus eliminating the "lost in hyperspace" problem.

Graphic ιools. Our graphic tools are in the midst of a generational change at this moment. Appendix B provides an account of where we are going, but it is also worthwhile to review where we have been recently. Appendix D provides more detailed technical information. The device models and the various possibilities for verbal and graphical coaching that map onto them posed a major data management problem for Sherlock II. Even experts' simplified representations of circuits were reasonably complex, and there were quite a large number of system components that Sherlock needed to know about. For any testable component in the simulation, it was necessary to define a graphic that would appear when the trainee wanted to test that component, along with "hot spots" in the graphic corresponding to test points. With such information, the system could interpret button presses on the mouse as pointing to the test points to which simulated meter probes should be attached. In addition, the simulation needed to be able to display a variety of control panels, from both the test station and the hand-held meters used for diagnosing test station failures. All in all, there were 1500 separate video images involved, with as many as 100 hot spots on a single image.

We thus faced interacting problems of simulating a complex system, providing an interactive interface to various parts of the system, and developing a database of all the views, hot spots, etc., involved in providing the simulation interface. Further, pragmatic considerations required that while most views of control panels and components were delivered via video, a few could not be photographed and had to be delivered via computer graphics. As a result, we needed the system to be deeply object-oriented, so that telling a component to display itself, for example, did not require knowing whether this involved drawing a computer graphic or requesting a particular frame from the videodisc.

To handle these various needs, we developed a variety of tools for internal project use. These tools were not meant for export, but they proved to be powerful enough to permit not only Sherlock II, but also a prototype for an entirely different tutor under development at Brooks AFB, to be built efficiently (the Brooks tutor used flowcharts as a basic data structure, and we were able to generalize our tools to create a flowchart drawing tool for inputting both the appearance and the meaning of flowcharts efficiently. For the Sherlock work, we built a tool that permitted inputting of information about front panels and components efficiently.

We have now generalized these tools into what Ted Hughes calls a hypergraphic editor. This rich specialization of an object-oriented graphics editor permits the efficient development of the databases needed to support simulations of complex systems. Appendix B provides an account of this improved capability. We expect to be able to distribute this tool within a few months, and a copy will be filed with DTIC.

Version control browser. Like all large software engineering efforts, Sherlock II generated large numbers of versions as different programmers added and debugged different system components. Version control for object-oriented systems is not yet a well-developed capability, so we had to develop some tools to help us manage the software development process. A version control browser has been built that deals with the core of the object-oriented software development process.

The core issue that we faced was controlling modifications in object definitions. In a mature project, there is a well-developed library of objects that form a base for the work. These objects are treated as givens in the system environment—they are seen as immutable. In a developing project, however, it is not always entirely clear at the outset how an object should be specified. Both the structure of its declarative knowledge and the specifics of its procedural methods may fluctuate. With parallel development going on, it becomes possible, therefore, for two different programmers to make conflicting changes in an object. Everyday version control programs might be able to flag parts of the code listing as being involved or not involved in a series of changes, but those approaches are not able to provide guidance that is organized at the object level rather than at the program text level.

Maria Gordin in our project developed a version browser to overcome this problem. This browser operates on what we call "project files." New parts of Sherlock are stored in these project files, and the Sherlock program image is developed by reading in, in turn, the various project files that comprise the system—some dozens of them by now. The version browser allows programmers to examine the contents, in terms of objects, of each project file. When cycles of changes are detected, i.e., when one file changes an object and a later file changes it again, perhaps in a conflicting way, the resulting cycle is flagged so that the lead programmer can check to see if there are any conflicts in the object redefinitions generated by the different files. Related display tools permit the lead programmer to reorganize objects from time to time to create project files that are more consistent and non-interacting.

Menu tools. In order to preserve the rich range of menu response capabilities we had in Sherlock I, we needed to re-invent some basic menu routines in a form related to that given us as part of the Xerox AI environment. In fact, we developed several basic resources. One was a multicolumn menu. Menu systems in most languages fail to consider the possibility that a large number of choices might need to be placed before a trainee. For example, sometimes a trainee using Sherlock must select from among hundreds of circuit components. We developed two tools to support this, pop-up menus, and multicolumn menus. Pop-up menus appear only when a specific choice is needed from a trainee. Among the possibilities they permit is that of homing in on a choice by progressive refinement. For example, a trainee using Sherlock II might ask to have a component displayed. A pop-up menu might then ask him which part of the test station holds the component. Subsequent pop-up menus home in further, ending with a choice among alternatives. For example, to display a particular circuit board labeled A1A3A15, a trainee might first ask for a display change. From a pop-up menu of system areas, he might select RAG (relay assembly group) as the part of the system on which he wants to focus. Then, he might see a second pop-up menu listing major types of system components. If he selects "card," he might then see a pop-up menu of major card categories in the RAG drawer. After picking one of the choices, he would be given a list of all cards matching the requirements just developed.

This last list might include a large number of choices. To fit neatly in a menu system, the choices might need to be shown in multiple columns. As a result, we added an object called a multi-column menu to support this requirement. At a deeper level, of course, the hypergraphics tools will support development of arbitrary displays with hot spots, so the possibilities for menu formats are endless. However, we find it useful to handle, as special simplified cases, permanent menus (that stay on the screen), pop-up menus, and multicolumn menus.

We also added one further capability, self-explaining menus. Borrowing a concept from the Xerox environment once again, we developed menus with the special property that if the mouse is held over a particular choice option for more than a certain interval (typically about 1.2-2 sec), a textbox appears describing the effect that will be generated if the mouse button is pushed while the mouse is over that option. This provides exactly the kind of help that we prefer. There is nothing to learn about. If one hesitates while deciding on a choice, the meaning of that choice is displayed. Otherwise, the system stays in the background.

Window tools for friendlier multi-pane and multi-window applications. We have, in order to support our work, developed a variety of tools that make both using and programming interfaces within Smalltalk more comfortable. These are described in Appendix B. While these are not, as such, tools that we would distribute on their own, programmers will find them useful. Hence, we are working hard to document and illustrate the uses of these tools as part of the overall hypergraphics capability we have developed.

The current generation: KeyPad. Appendix C describes the generalization and evolution of the menu concept into a generic tool that will be of much broader utility, since it includes not only standard text menus but also the increasingly popular buttons and even a variety of general "hot spot" schemes.

Formatting tools for text windows. We needed to extend the text window capability of Smalltalk in order to accommodate Sherlock II. This resulted in the development of an editing command processor for Smalltalk text windows. This processor, a sort of primitive version of the Scribe concept, allows text to be indented, colored, and otherwise manipulated as it is "poured" into windows. In fact, we have found it extremely useful to have parts of explanations colored. An obvious extension now being worked on is to have certain colored segments also active as "hot spots." This sort of tool is common in high-end work stations but will be more broadly accessible if available in systems that run on pc's.

Interactive videodisc interface. A final tool set that we created is for control of an interactive video system. These tools allow a videodisc player, currently a Sony system, to be controlled from within Smalltalk. This sounds like a small matter, but because of lack of source code for either the Sony drivers or the kernel of Smalltalk, it required considerable reverse engineering work. Related tools compensate for display "dirtiness" produced by the Sony drivers. The Sony drivers are needed to handle the mouse when their VGA GENLOCK card is used to combine computer graphics with a video frame (at present, the Sony VIEW hardware is the Air Force standard, at least within the Tactical Air Command).

Appendices

A. Katz and Lesgold: Modeling the Student in Sherlock II.

B. Hughes: Graphic Tools in Smalltalk/V.

C. Peters: KeyPad: A Versatile Interaction Device.

D. Peters: Graphic Editors: Technical Details.

Modeling the Student in Sherlock II

Sandra Katz and Alan Lesgold

August 30, 1991

Paper Submitted to the International Joint Conference on Artificial Intelligence
Workshop on Agent Modeling for Intelligent Interaction

Table of Contents

Modeling the Student in Sherlock II

Sandra Katz and Alan Lesgold

Learning Research and Development Center
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
United States of America
Phone: 1-412-624-7054
FAX: 1-412-624-9149
Electronic Mail: katz@unix.cis.pitt.edu

INTRODUCTION

Considerable debate surrounds the issue of whether techniques for student modeling should continue as a major agenda item for researchers and developers of computer-based instruction. Borrowing Lajoie and Derry's terminology for describing the two extremes of the debate spectrum (Lajoie and Derry, in press), "model builders" regard precision modeling as the key to individualized instruction, and believe that student performance can be modeled, traced, and corrected in the context of problem solving (e.g., Anderson et al., 1985). In the opposite camp, "model breakers" choose not to model student performance because they believe that it is neither technically feasible to do so (e.g., Soloway, 1990; Fox, 1988a, 1988b) nor pedagogically sound, since approaches like model tracing "force" the student down a solution path of the machine's choosing, instead of encouraging him or her to experiment with different solution paths and engage in self-diagnosis and correction.

One factor slowing down the resolution of this debate is that we lack cognitive models of the processes involved in human tutoring (Putnam, 1987, Chi & Bjork, in press). Several studies have been done to uncover the repertoire of techniques that experienced human tutors use, as well as the plan-based constraints which govern decisions about when to apply each technique (Lepper et al., in press; Lepper, 1989; Putnam, 1987; Reiser, 1989; Woolf & McDonald, 1984). Although this research has contributed much to our understanding of human tutoring, it has yielded some inconclusive and even inconsistent findings (McArthur et al., 1990, p. 232).

In particular, it is unclear to what extent human tutors construct detailed diagnostic models of their students' understanding. Putnam (1987) and McArthur et al. (1990) found no evidence that tutors try to uncover the exact nature of students' errors and what causes them, as would be consistent with what they call a *diagnostic/remedial model* of tutoring. Rather, tutors' primary goal seems to be to help students correct errors, so as to be able to move onwards through the curriculum.[1]

[1] Putnam (1987) and McArthur et al. (1990) differ in their accounts of the role that the curriculum plays in governing tutors' actions. Putnam suggests that tutors rather rigidly follow a curriculum script—"a loosely ordered but well defined set of skills and concepts students [are] expected to learn, along with the activities and strategies for teaching this material" (p. 13). However, McArthur et al. observed that teachers adapt the global plans of such a script opportunistically, based on events in the tutorial situation.

However, these researchers qualify their findings, acknowledging that the experienced tutors they observed might have been so familiar with the subject matter they were teaching, the students who they were teaching, and/or the difficulties that students commonly experience, that they did not *need* to overtly probe students' understanding. Student modeling by direct pattern recognition is still student modeling.

Further doubt on the validity of dismissing the diagnostic/remedial model of tutoring offhand is cast by other researchers' (Lepper, 1989) observation that tutors distinguish between "disruptive" and "productive" errors, interrupting students during problem-solving sessions only to correct the former, while waiting until after the student has solved the problem to address errors from which the student could learn something. One wonders how tutors could make this distinction without *recognizing the exact nature of students' errors* to begin with. It thus seems that the real question we need to address is not *if* a diagnostic/remedial model of tutoring is correct, but *when* it applies and when it does not. "Model breakers" would argue that until we know for certain if, how, and/or when human tutors do deep modeling of students' understanding, we should not attempt to build student modeling components.

Somewhere in the middle ground between "model-builders" and "model-breakers" stand researchers who believe that student modeling components can be built which support the computerized tutor's role as mentor in apprenticeship-style training (Lajoie and Lesgold, 1990), without interfering with the opportunities that the computerized learning environment affords students for experimentation, reflection, and self-diagnosis. Instead of attempting to construct precise models of students' performance and understanding, these researchers aim to capture human teachers' tendency to formulate imprecise, qualitative assessments of students' ability (Hawkes et. al., 1990; Lesgold et. al., in press), as manifested in statements like "Joe is *fairly good* at *single-column subtraction*, but *weak* at *borrowing across columns*." In short, these researchers take a conservative stance towards student modeling; they model students to the degree that we can be certain that human tutors do, using techniques that we are sure that human tutors use—namely, by taking students' behaviors as indicators of what they do or do not know.

This is, of course, a non-trivial task—for humans as well as for machines. Several writers who have discussed the difficulties involved with modeling students (e.g., Chin, 1989; McCalla & Greer, 1990) have emphasized the problem of handling uncertainty about what students do and do not know. There is often more than one possible explanation for students' inappropriate actions and errors. Each error could be traced to several conceptual misunderstandings; each inappropriate action to various skill deficiencies. So, ambiguity is a major problem in student modeling. Add to this idiosyncratic errors, such as computational (e.g., addition) or mechanical (e.g., typos) slip-ups, and forgetting prior knowledge and the problem compounds. McCalla & Greer (1989) describe several recent approaches to handling uncertainty in interpreting student actions, such as bounded student models (Elsom-Cooke, 1989) and granularity-based reasoning (McCalla & Greer, 1988).

Another approach to handling uncertainty about student knowledge was proposed by Sharon Derry and her colleagues at Florida State University (e.g., Hawkes, Derry & Kandel, in press). The approach is derived from fuzzy set theory (Zadeh, 1965), a mathematical theory that attempts to capture the notion that items can have varying degrees of membership within a set. For example, students can have partial membership within the set of people who are expert on a particular skill. Hawkes et al. (1990) present the following rationale for applying fuzzy set theory to student modeling:

> [Partial membership within a set] is an important concept to the field of 'TSs [intelligent tutor systems] because there are different aspects of vagueness inherent in real world data. There is the inherent vagueness of classification terms referring to a continuous scale, the uncertainty of linguistic terms such as "I almost agree," or the vagueness of terms or concepts due to statistical variability in communication (Zemankova & Kandel, 1984). Fuzzy set theory is an attempt to provide a systematic and sound basis for the modeling of [those] types of imprecision which are mainly due to a lack of well-defined boundaries for elements belonging to the set of objects. The use of fuzzy terms [eg., "rather high," "possibly," "not likely," etc.] allows for imprecision and vagueness in the values stored in the database. This provides a flexible and realistic representation that easily captures the way in which the human tutor might evaluate a student...Also, many tutoring decisions are not clearcut ones and the capability to deal with such imprecision is a definite enhancement to ITS's. (pp. 416-17)

At the University of Pittsburgh's Learning Research and Development Center, we have been cooperating with Derry and her colleagues in developing techniques to implement this imprecise modeling approach, which we will refer to as *fuzzy student modeling*, or FSM. We have built a prototype FSM component for Sherlock II (Lesgold, Eggan et al., in press), an intelligent coached practice environment, which is being developed to train Air Force technicians to troubleshoot faults in a complex electronic testing system. Our primary aims in developing the system's prototype modeling component were (1) to identify the information that the student modeling knowledge base needs in order to build an imprecise model of a student, and (2) to formulate algorithms for building and updating a student record. Correspondingly, our main goal in this paper is to describe what we have learned about the knowledge and software engineering tasks involved in building an FSM component, so that others who wish to do so may incorporate fuzzy modeling techniques within their system.

Currently, the prototype FSM in Sherlock II is complete, and the system is being field tested through this summer. Lacking sufficient data on actual student performance to guide the development of the prototype student modeling knowledge base—in particular, the rules for updating a student record—we have no illusions that the information contained within the knowledge base is correct. Consequently, one of the principal goals of field testing is to gather data (in the form of traces of student problem-solving sessions, and responses to interviews) which can be used to fine-tune the student modeling knowledge base and updating algorithms. Thus, an additional function of this paper is to describe some of the techniques we are considering for fine-tuning the student modeling component so that it works as it should.

Overview of paper. After an introduction to Sherlock II, we describe the student modeling component in more detail, from both a knowledge engineering and software engineering perspective. We specify what knowledge is represented in the student modeling component, how this knowledge is represented, and how an individual student model is acquired and updated. We then propose a framework for evaluating the student modeling component in an intelligent tutoring system (ITS). Applying this framework to the prototype FSM component in Sherlock II, we discuss how we plan to go about revising the system's student modeling knowledge base and updating procedures. Finally, we summarize what we have learned about FSM, and student modeling in general, from the work described here.

THE FUZZY STUDENT MODELING COMPONENT IN SHERLOCK II

Overview of Sherlock II[2]

Sherlock II is a coached practice environment developed to train avionics technicians to troubleshoot a complex electronic testing system—in particular, a test station that checks out aircraft modules. Ordinarily, when a module from an F-15 aircraft is brought into the repair shop because of suspected malfunction, the technician attaches the module ("unit under test" or UUT) to a *test station*, and, by carrying out a series of test procedures, is able to locate the fault. However, sometimes the airman discovers that he[3] continues to get an unexpected reading even when he replaces the UUT with a shop standard. When this occurs, the airman should know that the problem is not with the UUT; rather, the test station itself is malfunctioning and needs to be repaired. Locating a faulty component within the test station requires the technician to explore a much larger problem space than does troubleshooting a faulty module; the test station contains approximately $70ft^3$ of circuitry. However, not all of this circuitry is needed to carry out each checkout procedure. So, the essential task confronting the airman is to construct a mental representation of the active circuitry—what we refer to as the *active circuit path*—and to troubleshoot this path until he locates the fault. Sherlock's job is to scaffold the process of learning to construct these abstract representations and to develop effective troubleshooting strategies.

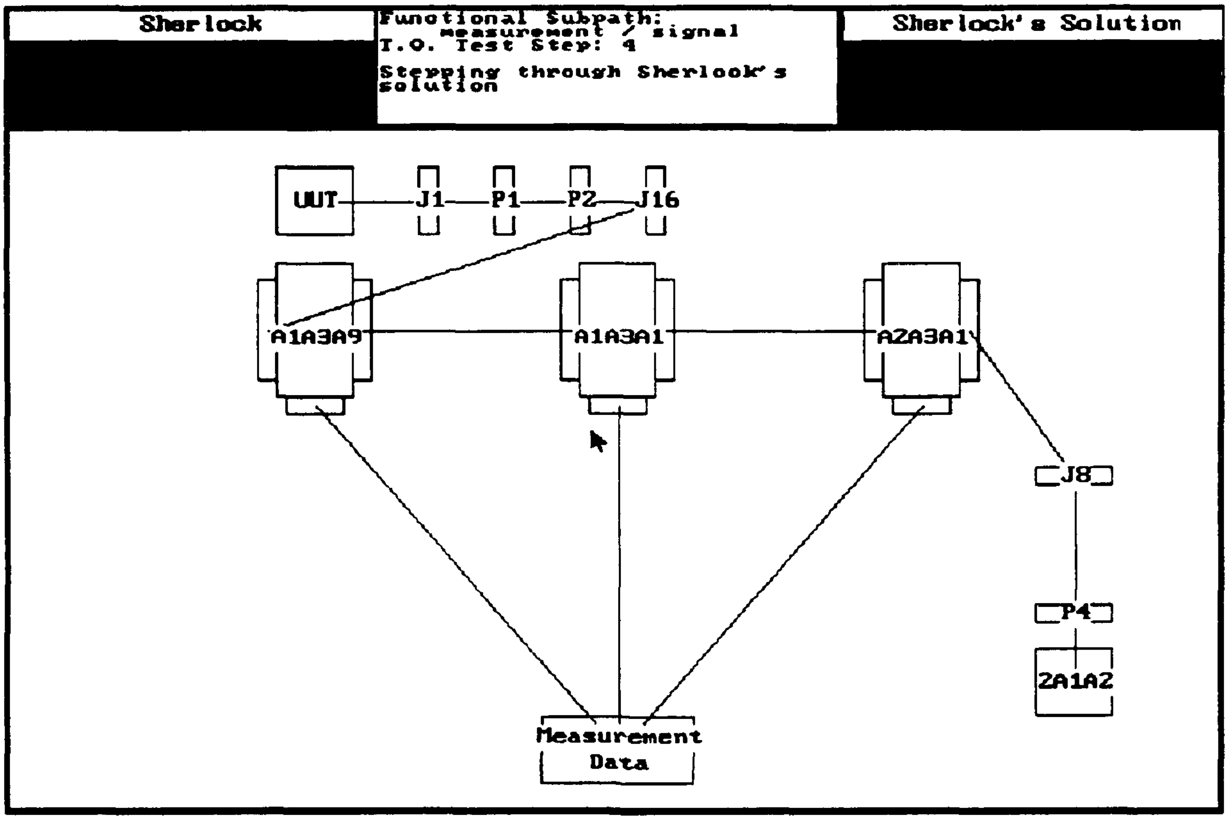
Sherlock II is a realistic computer simulation of the actual job environment. Trainees acquire and practice skills in a context similar to the real context in which they will be used. (The system is written in Smalltalk/V286, an object-oriented programming language, and runs on 386 workstations equipped with Sony videodisc hardware.) Sherlock presents students with a series of exercises of increasing complexity. There are two main episodes in each Sherlock II exercise: *problem-solving* and *review*. During problem-solving, the student runs a series of checkout procedures the aircraft unit suspected of malfunction. Using interactive video, the student can set switches and adjust knobs and dials on test station drawers, take measurements, and view readings. If he gets an unexpected reading on one of the measurement devices (i.e., a handheld meter, digital multimeter, or oscilloscope), he should first see if the UUT is the culprit by replacing it with a shop standard. If after doing this he still gets an unexpected reading, the student should troubleshoot the test station. He can test components by attaching probes to pins on a video display of an extender board, replace a suspect component with a shop standard and rerun the failed checkout test, etc. These and other options are available through the **Troubleshooting Menu**.

Perhaps most importantly, the student can ask for help at any point while troubleshooting. Sherlock provides help at both the circuit path and individual component levels of analysis. Within each level, Sherlock offers functional/conceptual ("how it works") help, and strategic/procedural ("how to test") help. For example, if a student asks for strategic help at the circuit level, he will first be told which troubleshooting goals he has accomplished, in terms of the significant functional areas of the device, such as, *You have verified the stimulus side of the test station.* Similarly,

[2] More detailed discussions of the Sherlock tutors can be found in (Lajoie & Lesgold, 1989; Lesgold, in press; Lesgold, Eggan et al., in press; Lesgold, Lajoie, et al., in press).

[3] We use the masculine pronoun, since about three-fourths of our students are male.

if a student asks for help with testing a component, he will first be told which component-testing goals he has achieved, such as, *You have verified that the inputs to the A1A2A3 card are ok.*



Sherlock

Functional Subpath:
measurement / signal
T.O. Test Step: 4

Stepping through Sherlock's solution

Sherlock's Solution

Figure 1: Example of Circuit Path Display

Coaching becomes increasingly directive if the student asks for repeated help, and is guided by the system's model of the student. If the student model suggests that the student should be able to make the next measurement with only a little help, Sherlock will give minimal support, thereby encouraging the student to interpret the hint himself, and to try to come up with the next measurement to make on his own before requesting help again. However, if the student model suggests that the student will have difficulty (e.g., he has had trouble testing a particular type of component before), Sherlock will give several levels of help right away. More directive hints recommend a goal to achieve next, and—at the furthest extreme—information about how to achieve the next goal (e.g., by specifying which pins to test, on a particular component). At certain stages of circuit-level hinting, the student is shown a diagram of the active circuit path, as shown in Figure 1, color-coded to indicate which circuit testing goals have been achieved, or remain to be achieved. With this coaching scheme, success in solving the problem is assured, and the student is largely responsible for his own learning.

While the student is troubleshooting, Sherlock creates a record of the goals the student has achieved, and how he achieved them, at both levels of analysis (circuit and component). This *goal/test record* contains a trace of the student's solution, and is the crucial input to the second "episode" of a Sherlock II exercise, review, or what we call "reflective follow-up." (The menu for *reflective* follow-up is shown in Figure 2.) In this phase, a trainee can replay his problem solution step by step. At each step, Sherlock provides an evaluation of what the trainee did—flagging each action as appropriate (with a green check) or inappropriate (with a red cross)—and briefly tells the student *why* an action was flagged as inappropriate. A number of information sources are available to the trainee as well. For example, he can ask what an expert would have done instead. Also, he can examine simplified diagrams of the circuitry which are organized to provide good representations of the problem, as shown in Figure 1. We are working on additional coaching to explain why the diagrams are structured as they are. The drawings are interactive: mousing on any drawing component produces an explanation of what is known about the status of that component given the actions (such as measurements on the circuit) carried out so far. Finally, in addition to these opportunities for reviewing one's own performance, there is also the opportunity to simply run through an expert's solution to the problem, with the same informational resources available at each step of the expert solution.

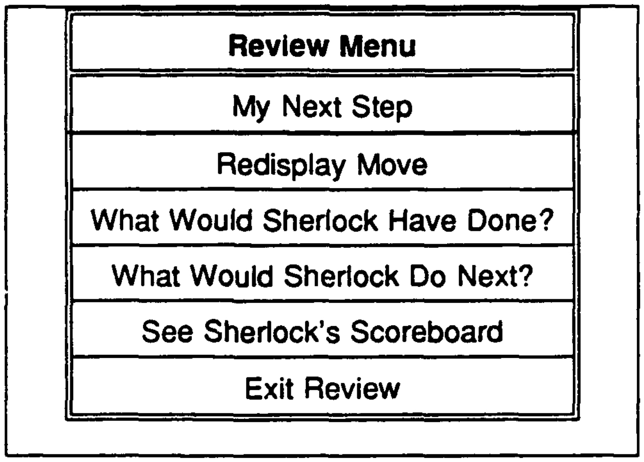

Review Menu
My Next Step
Redisplay Move
What Would Sherlock Have Done?
What Would Sherlock Do Next?
See Sherlock's Scoreboard
Exit Review

Figure 2: Menu for Reflective Follow-up

Theoretical Basis for the Modeling Component

A recurring theme in the literature on user modeling in general[4] is that an important distinction can be drawn between facts about a user, and information that is actually *used* by the system for decision-making. For example, Sparck Jones (1989, p. 345) distinguishes between *non-decision* and *decision* properties of a user, which must be categorized in relation to the function of the system. An example of the former, in, say, a voter advisor system, would be dress color preference; an example of the latter would be political belief. Ohlsson (1986) draws a similar distinction with respect to cognitive diagnosis in intelligent tutoring systems. He contrasts diagnostic approaches that attempt to "explain, or account for, a set of observations (performances)" and those that, in addition, "take into account how the diagnosis they produce is going to be *used*" by the system, for instructional purposes. The former approach could be called "research in miniature," (Ginsburg, 1983); the latter, according to Ohlsson, is *pragmatic diagnosis*.

A similar distinction manifests itself in the curriculum theory that underlies the approach to student modeling taken in Sherlock II.[5] Lesgold (1988) has argued that more attention should be paid in instructional design to the difference between the pieces of knowledge (i.e., concepts and skills)

[4] Following others (eg., Gegg-Harrison, 1990), we use "user modeling" as a broad term which covers building user models in natural language systems, and intelligent systems in general.

[5] See Lesgold, Eggan et al. (in press) for a discussion of other theoretical underpinnings for the approach.

that constitute a subject domain on the one hand, and those knowledge components that are actually required for expert performance in that domain on the other. He drew this distinction in response to two related observations: (1) that skilled performance does not always depend upon knowledge of all of the facts, concepts, etc. associated with a domain (e.g., a good electrician might not know all of the laws of electricity), and (2) that a good grasp of all of the knowledge components making up a domain does not necessarily guarantee skilled performance (e.g., the "A" student in electronics will not necessarily be a good electronics technician; the medical student who can recite all of the symptoms associated with a disease will not necessarily be able to diagnose patients who have that disease). The latter phenomenon can be explained by the fact that students often lack the "glue" that holds distinct knowledge components together; i.e., the links that make the conceptual whole more than the sum of its parts.

In light of these observations, Lesgold proposed a three-tiered model of instructional knowledge for an intelligent tutor. This model can be represented as a lattice, since nodes may be related to more than one higher-level node. To summarize, the lowest levels of the knowledge lattice consist of skills and concepts required for expertise in the domain. Increasingly higher levels of knowledge in this tier (called the "Knowledge Layer") consist of aggregations of these expert knowledge components. Each aggregation and, more importantly, the "glue" that holds its pieces together, constitutes an instructional goal. The next tier, the "Curriculum Layer," organizes these hierarchies of knowledge aggregations according to a particular perspective, each perspective defining a curriculum or "goal structure for instruction" (p. 123). Finally, sitting above the Curriculum Layer on the third tier are Metaissues—e.g., considerations such as a student's aptitude, learning style, etc.—which guide the system's decisions about which of the various curriculum perspectives should be in focus at any given time.

Thus, in this three-tiered theory of instructional knowledge, there is a distinction between *knowledge components* and *curriculum goals* which mirrors the distinctions between descriptive and pragmatic diagnostic knowledge, and non-decision/decision properties of a user discussed in the user modeling literature. As we will demonstrate below, these ideas have informed our design of the student modeling knowledge base for Sherlock II. They have also shaped our thoughts about the different viewpoints from which the student modeling component of an ITS can be assessed.

Overview of the FSM Approach in Sherlock II

The main structure used to represent instructional and diagnostic/modeling knowledge in Sherlock II is the *student variable lattice*. Each node in the lattice is an indicator about some characteristic of student capability. The lowest level variables, which we call *local variables* (e.g., *ability to use the digital multimeter*), represent distinct pieces of expert knowledge. The higher-level variables, which we call *global variables*, represent abstractions over groups of these "local" competencies (e.g., *ability to use test equipment*, which includes the digital multimeter, handheld meter, and oscilloscope). Global variables, moreso than the local variables which they subsume, reflect expert judgments about what the indicators of expertise in a given domain are, and, consequently, what the goals of instruction should be.

Currently, all student modeling variables are represented as *fuzzy variables*. A fuzzy variable can be thought of as a distribution over the set of possible levels of competence or knowledge a trainee might have in a particular skill. Local variables are updated by rules that fire when certain performance conditions have been met (e.g., student replaces a component before fully testing it). The updating rules for a local variable specify when to change the current distribution, how much to change it, and in which direction (upwards or downwards). A sample fuzzy variable for a local variable, including its updating rules, is shown in Table I. Changes in local variables percolate upwards through the variable network, thereby updating global variables. Equations containing weighted combinations of local variables are used to update global variables. (More will be said about fuzzy variables and their updating rules below.)

Currently, Sherlock uses a particular student model mainly to drive coaching and problem selection. The student's scores on selected variables determine how detailed a hint should be when the student asks for help, as well as how difficult the next problem should be. In addition, the student model is used to enable trainees to view a graphical display of their progress on particular variables, as shown in Figure 3.[6] We have also been exploring ways of using the student modeling component to support collaborative learning activities—for example, to intelligently pair students to work together on problems or critique other students' solutions (Katz & Lesgold, in press).

In light of this overview of the student modeling component in Sherlock II, we can take a closer look at the main components of the student modeling knowledge base. Then, we will consider how these components work together in building and updating a student model.

The Student Modeling Knowledge Base

The student modeling knowledge base in Sherlock II consists of two main components: the student variable lattice, described above, and rules for updating variables at the lowest layers of this lattice, the local variables.

The variable lattice. Local variables represent conceptual knowledge (e.g., *understanding of how a relay card works*), skills (e.g., *ability to use an oscilloscope*), behavioral dispositions (e.g., *tendency to replace components rather than test them*), and affective factors such as *motivation.* They are thus close to an overlay of competence estimates on the system's expert model, which corresponds to the Knowledge Layer in the model of instructional knowledge described above. Global variables represent aggregations of these knowledge components, such as *troubleshooting strategy*, domain knowledge (e.g., *knowledge of how the test system works*, as a whole), and *test equipment usage.* Thus, global variables represent an overlay on the Curriculum Layer of instructional knowledge. Table II contains a list of the main types of student variables tracked by the modeling component in Sherlock II.[7]

[6] The two scales corresponding to each variable represent the student's score up to the current problem (old) and including the current problem (new). The poles of each scale represent student ability levels—novice (N), journeyman (J), and expert (E). Note that the student's score on *systematicity* has decreased substantially in the current session; a new scale (from novice to journeyman, instead of from novice to expert) was required to show this.

[7] Dotted items correspond roughly to global variables; indented items to local variables.

Table I. Upgrading Conditions for One Fuzzy Variable.

Example of a Sherlock Fuzzy Variable and its
Upgrading and Downgrading Conditions

HANDHELD METER USAGE

description: measure of knowledge about when to use the handheld meter; tactical ability is assumed

initial distribution: journeyman

upgrading conditions:

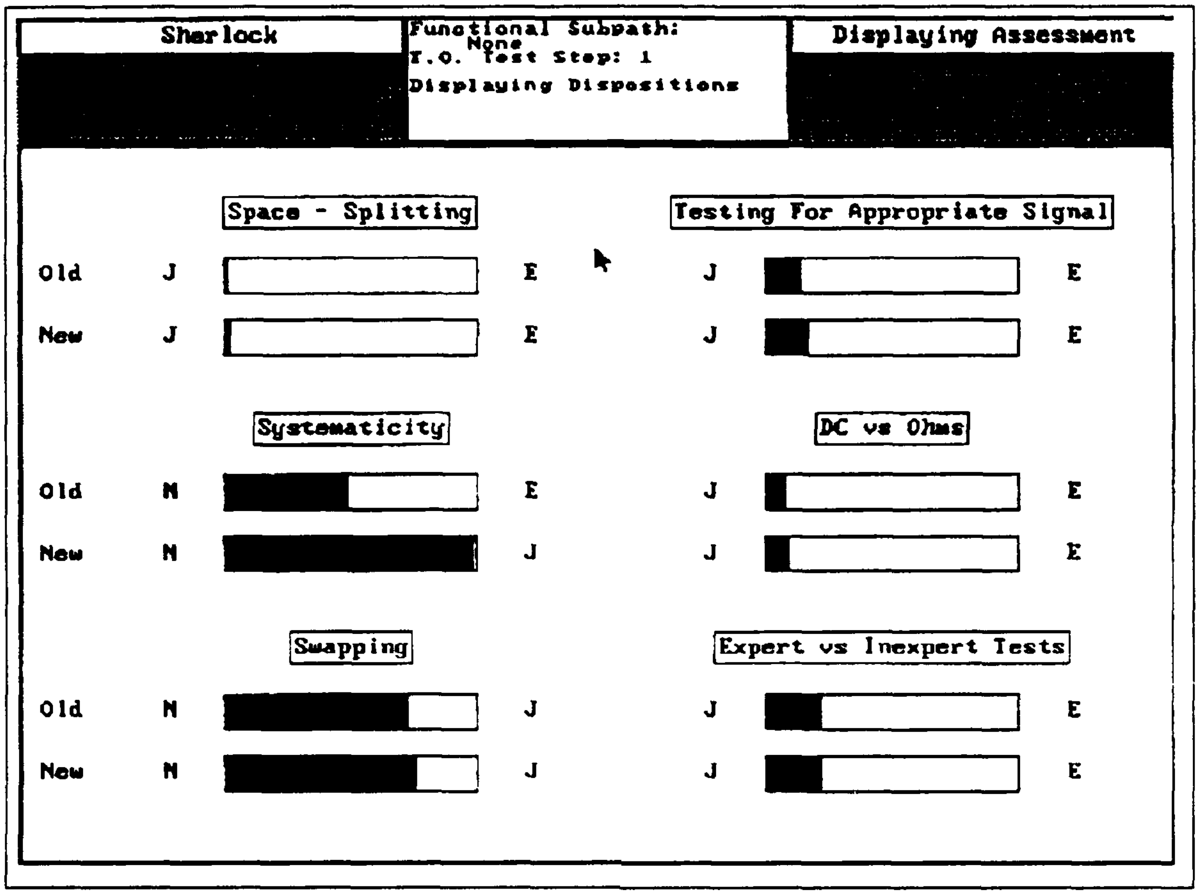
1) uses handheld meter appropriately to measure resistance, i.e., when there is no power, without help from Sherlock (+++)
2) uses handheld meter appropriately to measure DC voltage, i.e., DC power is on, without help from Sherlock (+++)
3) uses handheld meter appropriately to measure AC voltage, i.e., AC power is on (+++)

downgrading conditions:

1) uses handheld meter to measure resistance when power is on (e.g., shorting in the data area) (---)
2) uses handheld meter to measure voltage when there is none (---)
3) uses handheld meter to measure DC voltage when it's AC (--)
4) uses handheld meter to measure AC voltage when it's DC (--)

Where do these variables come from? In developing the prototype modeling component in Sherlock II, we relied upon two main sources of information about which aspects of students' understanding and performance should be modeled by the system: cognitive task analysis, and expert judgments. As we have described elsewhere (Lesgold, Eggan et al., in press), local variables, and the rules for updating them, are derived mainly from observable properties of troubleshooting performance, as revealed during empirical analyses of expertise. To a lesser degree, local variables reflect expert judgments of what properties of local performance are important to measure. Global variables, on the other hand, are anchored primarily in expert evaluations of trainee performance.[8] Indirectly, global variables are also anchored in observations of human performance, since they represent cumulations of local variables. These relationships

[8] Policy-capturing techniques (Lesgold, in press; Nichols et al., in press) can be used to identify scoreable properties of student performance traces. The set of evaluation criteria identified by using these techniques keeps the amount of sensible cumulations of local variables to a tractable size.



Sherlock

Functional Subpath:
None
T.O. Test Step: 1
Displaying Dispositions

Displaying Assessment

Figure 3: Progress Report

between modeling variables and their sources are summarized in Figure 4.[9]

Representation of modeling variables. As we remarked above, one distinctive feature of our approach is the use of fuzzy variables, or competence distributions, rather than scalar quantities as the primary structure for representing student modeling information.[10] We have five such knowledge states for each student modeling variable: *no knowledge, limited knowledge, unautomated knowledge, partially automated knowledge,* and *fully developed knowledge.* Initially, if we know absolutely nothing about a trainee, we might assume that each of the five states has equal (20%) probability, as shown in the top part of Figure 5. If we have some prior knowledge, we can bias the distribution, of course. For example, if we know that the fuzzy variable indexes a particularly hard skill, we might specify the distribution as (20%, 60%, 20%, 0% 0%). This would indicate that we are 60% certain that the knowledge indexed by this particular variable is limited in this particular student, but it might be non-existent, and it might even have reached the level of being established but not automated. The distribution of fuzzy variable *i* can be denoted by the vector F_i, with the *j*th probability category of F_i being denoted by F_{ij}. So, if the top of

[9] Solid lines represent primary sources of information about variables; dotted lines represent secondary sources.

[10] For decision-making purposes, distributions are converted to scalar values.

Figure 5 represents F_i, then F_{i2}=0.60.

Updating rules for local variables. What a teacher (human or machine) can know about what a student knows is conveyed through the student's actions, including both speech acts and non-linguistic actions. Indeed, no agent modeling system can peer directly into the agent's mind; language and action are the medium through which modeling information passes, and there will invariably be gaps and distortions in the image conveyed. This is, in essence, why the modeling task is so hard. In a system like Sherlock II, that does not have natural-language processing capabilities,[11] student actions serve as the sole "window" or "keyhole" (Cohen, Perrault & Allen, 1981) through which diagnostic information about a student is conveyed.

As with human tutoring situations, not all student actions observable by the computer tutor are significant. Just as a human tutor for arithmetic would probably not attend much to the fact that a student coughed or writes his 7's like 1's, the student modeling component of an intelligent tutor need not record *all* actions that a student makes via the system's interface. Only a subset of these actions are significant, from a diagnostic standpoint. And they are significant only when certain constraints are met.

For example, in our tutor, the student can select a menu option which will allow him to test a component. But the student's choice of the **Test Component** option is only interesting in light of the testing situation and other

Table II: Student Modeling Variables Currently Being Used

- Global testing ability variables
- Circuit variables
- Circuit strategy variables by path type
- Circuit tactical variables by path type
- Component variables: strategic
- Component variables: tactical
- Overall score on testing component *(strategic plus tactical ability)*
- Test equipment usage skills
- Other test-taking skills
    - overall ability to interpret test results
        - circuit-level ability to interpret test results
        - component-level ability to interpret test results
    - ability to read schematics
- Domain knowledge
    - system understanding
    - TO understanding
- Dispositions
    - swapping vs testing
    - testing for the appropriate signal type
    - thrashing
    - history-taking
    - overall systematicity and completeness
    - attention to safety preconditions
    - redundant testing
    - attention to probability of failure
    - independence and self-confidence
    - accepting help

[11] One of our colleagues, Johanna Moore, is currently working on building dialogue capabilities within Sherlock's reflective follow-up mode, using ideas derived from her dissertation (Moore, 1989). The goal of her work is to enable trainees to ask follow-up questions about the comments that the system generates while replaying the student's (or an expert's) solution. Eventually, similar dialogue capabilities will be incorporated within the problem-solving mode, thus enabling the student to ask follow-up questions about Sherlock's hints. Kass (1990) has commented upon the importance of flexible interaction between the system and the user for building the user model.

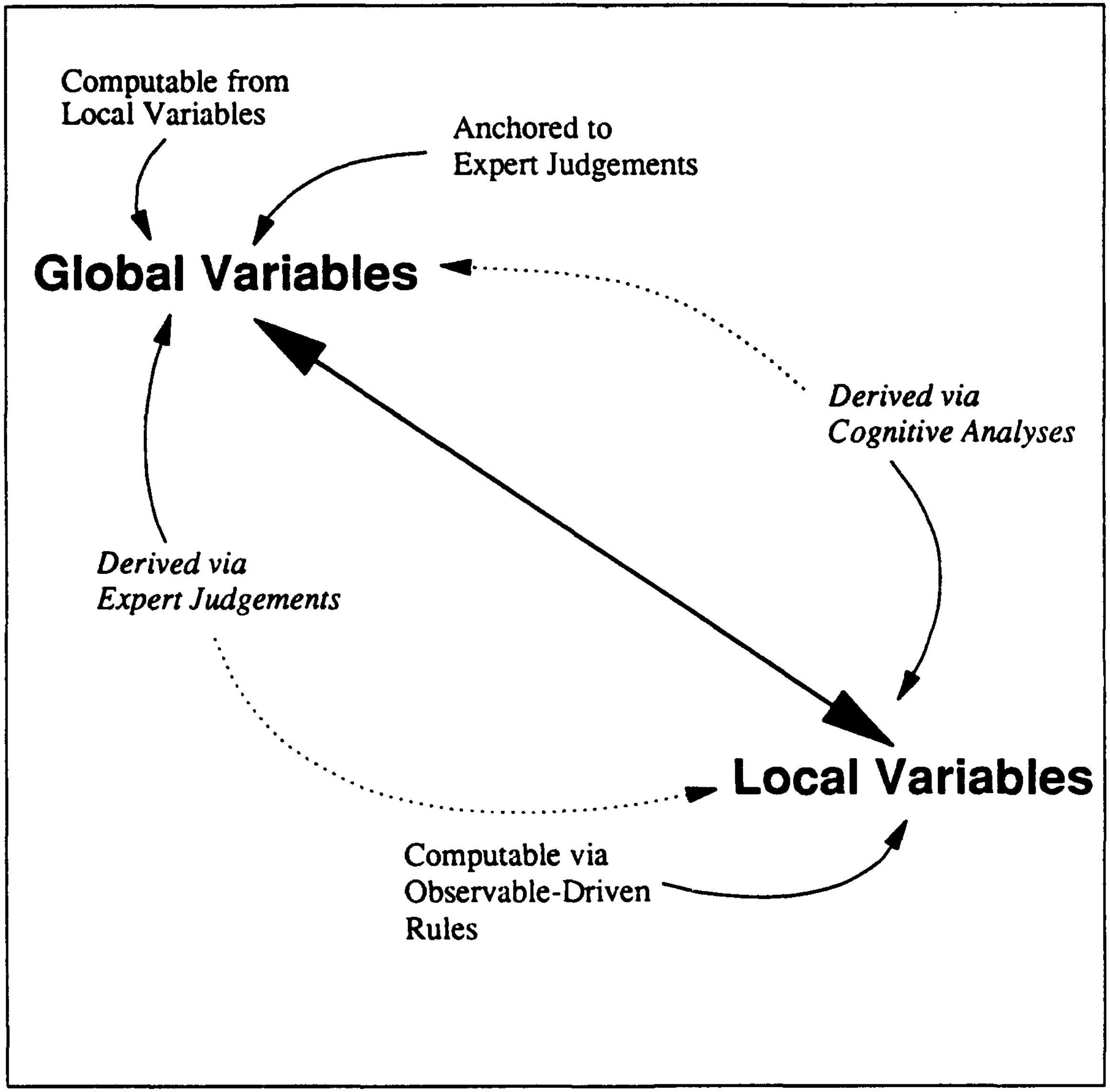


Figure 4. Multi-Level Assessment Scheme.

actions he has made. Is the component on the active circuit path? If not, then the student might have a weak model of the diagnostic test that the test station is currently performing, or of the test station itself. Has the student tested the component already, during the same problem-solving session? If so, then the student is carrying out a redundant test, and his overall troubleshooting approach might be inefficient and unsystematic. Did other tests made on other components already verify that the component is ok? If so, then perhaps the student has difficulty interpreting test results. In each case, we hedge our inferences because one behavioral event does not a complete diagnostic picture paint. Idiosyncratic behavior on the part of students, and multiple possible causes for student actions force us to use caution in drawing conclusions.

Thus, we once again revisit the distinction between data and information, decision and non-decision properties of users, etc., only at a more microscopic level than we have dealt with while discussing local and global modeling variables. In fact, we can view local variables as abstractions over diagnostically significant clusters of student actions, just as global variables are meaningful combinations of local variables, which address particular curriculum viewpoints.

In the Sherlock II context, we refer to these 'diagnostically significant' aspects of student behavior as *performance conditions*. Performance conditions comprise the "lefthand side" of rules for updating local variables—the events that trigger the system to change the current distribution for a fuzzy variable. The cluster of performance conditions that are used to update the fuzzy variable representing the student's ability to use a handheld meter is shown in Table I. Although the updating rules shown in this example are derived from simple observations (e.g., what test type the student set the meter to, voltage or ohms), not all local variables need to be fixed by simple events. Rather, some may have their values inferred by relatively complex qualitative patterns over multiple observable events. For instance, the local variable for the student's strategy in testing a particular functional area of the test station (e.g., the signal path between the UUT and the digital multimeter) may operate on implications of actions, such as an index of the extent to which a *series* of measurements on components within this functional area helped to identify the locus of the fault.[12]

Perhaps a more important observation to be made about performance conditions is that they are redundant across variables. The mapping of performance conditions to local variables in Sherlock II is many-to-many; several conditions are indicators for each variable, and a condition can be an indicator for more than one variable. This is not surprising, since in reality there could be more than one valid explanation for a student action. An action can be part of differing plans and/or motivated by different aspects of conceptual understanding. Referring to the example discussed in the preceding paragraph, an incorrect setting on the handheld meter—e.g., ohms instead of voltage—could mean that the student does not know how to set the meter up properly, from a purely mechanical standpoint, or that one needs to set the meter up correctly before using it.[13] What is more likely, though, is that the student does not know that the correct setting for the test he is conducting is VDC (direct current voltage), because he does not realize that current was in the active circuit path for the diagnostic test that the test station was carrying out when failure occurred. In other words, the student probably has an inadequate mental model of the failed checkout test. This aspect of domain knowledge is captured in two additional local variables besides *handheld meter usage*: *understanding of the Technical Orders*[14] and *ability to identify and conduct the appropriate test type* (DC vs Ohms).

In our system, we handle the uncertainty in interpreting student actions in two ways. First, we

[12] See Lesgold, Eggan et al. (in press) for additional examples of performance conditions in updating rules for local variables.

[13] This situation provides a good example of how violations of expected behavior can serve as indicators of gaps in students' knowledge. Kass (1990) stipulated a general rule for acquiring information about an agent called the Sufficiency Rule. The rule is based upon Grice's principles of cooperative action. *The rule essentially states that a cooperative agent does everything necessary (and nothing more) to enable the system to achieve its goal. In a tutorial situation,* it is the agent, not the system who has the goal of solving the problem, but we can assume that the Sufficiency Rule still applies—i.e., we should expect that the student will do all that is required to solve the problem (e.g., find the fault in the active circuitry). If he does not, it is probably because of a lack of understanding. We return to this point later.

[14] Technical Orders are a set of procedures that the technician follows to check a suspect module, the *UUT*, via the test station.

simply accept redundancy as an artifact of reality; thus, the same performance condition can appear in the updating rules for *several* local variables. Second, because performance conditions are stronger indicators for some of these variables than for others, we vary the *degree* or rate to which updating occurs.

In our system, rate of update is also a function of how much opportunity there is for adjustment, since events can happen with more or less frequency during a problem-solving session. If the same upgrade amount were used for all events, some variables would reach expert level way ahead of others. For example, there are many more opportunities to set the handheld meter, to place probes on pins, and to carry out other test-taking actions on a component than there are to respond to an indicator light which fails to come on. Identifying updating amounts that take both factors into account—interpretive strength of the behavior, and frequency of occurrence—is indeed a tricky matter, and we make no claims about having conquered it in our prototype. Below, we will describe some of the techniques we used to control updating rates in the prototype modeling component, as well as others we will try in building future versions.

The updating amount is expressed in the "righthand" side of an updating rule—the "action" part, in conventional terms. In the example shown in Table I, we use an arbitrary convention, where +++ means to upgrade a variable by a relatively large amount, ++ means a smaller upgrade, + means a still smaller upgrade, --- means a large downgrade, etc. An updating procedure for a fuzzy variable can be specified by two pieces of information, a range and a change percentage (which can be positive or negative, for an upgrade or downgrade, respectively). The range indicates what part of the fuzzy distribution, in terms of expertise levels, we should focus on,[15] and the change percentage indicates the magnitude and direction (up or down) of change we want to make within that focus. For example, if a trainee demonstrates incomplete knowledge of the troubleshooting procedure for a component *i*, then the update rule might specify a -10% change for (0%, 30%, 100%, 100%, 100%), which means downgrade the top three categories by 10% (1.00 × -0.10) and the limited knowledge category by 3% (0.30 × -0.10). The result, illustrated in the bottom part of Figure 5, would be a new F_i computed as follows:

$$\begin{aligned}
F_{i1} &\leftarrow F_{i1} + 0.10 \times 0.30 \times F_{i2} \\
F_{i2} &\leftarrow F_{i2} + 0.10 \times 1.00 \times F_{i3} - 0.10 \times 0.30 \times F_{i2} \\
F_{i3} &\leftarrow F_{i3} + 0.10 \times 1.00 \times F_{i4} - 0.10 \times 1.00 \times F_{i3} \\
F_{i4} &\leftarrow F_{i4} + 0.10 \times 1.00 \times F_{i5} - 0.10 \times 1.00 \times F_{i4} \\
F_{i5} &\leftarrow F_{i5} - 0.10 \times 1.00 \times F_{i5}
\end{aligned}$$

Note that this formulation automatically preserves the fuzzy variable as a distribution summing to 1.

Updating rules for global variables. As already mentioned, updating rules for global variables are expressed as weighted equations. Just as the change rates in updating rules for local variables reflect a performance condition's strength as an indicator for that variable, so the weights in aggregation equations reflect the relative strength of each local knowledge component (skill,

[15] In our system, we vary the range according to the student's current level of expertise—novice, journeyman, or expert. The percentage for each interval in the range decreases as skill increases. For example, the range for journeyman might be (0, 20, 30, 30, 30), for expert (0, 15, 25, 25, 25). This approach enables us to control updating so that the system does not deem the student an expert too readily.

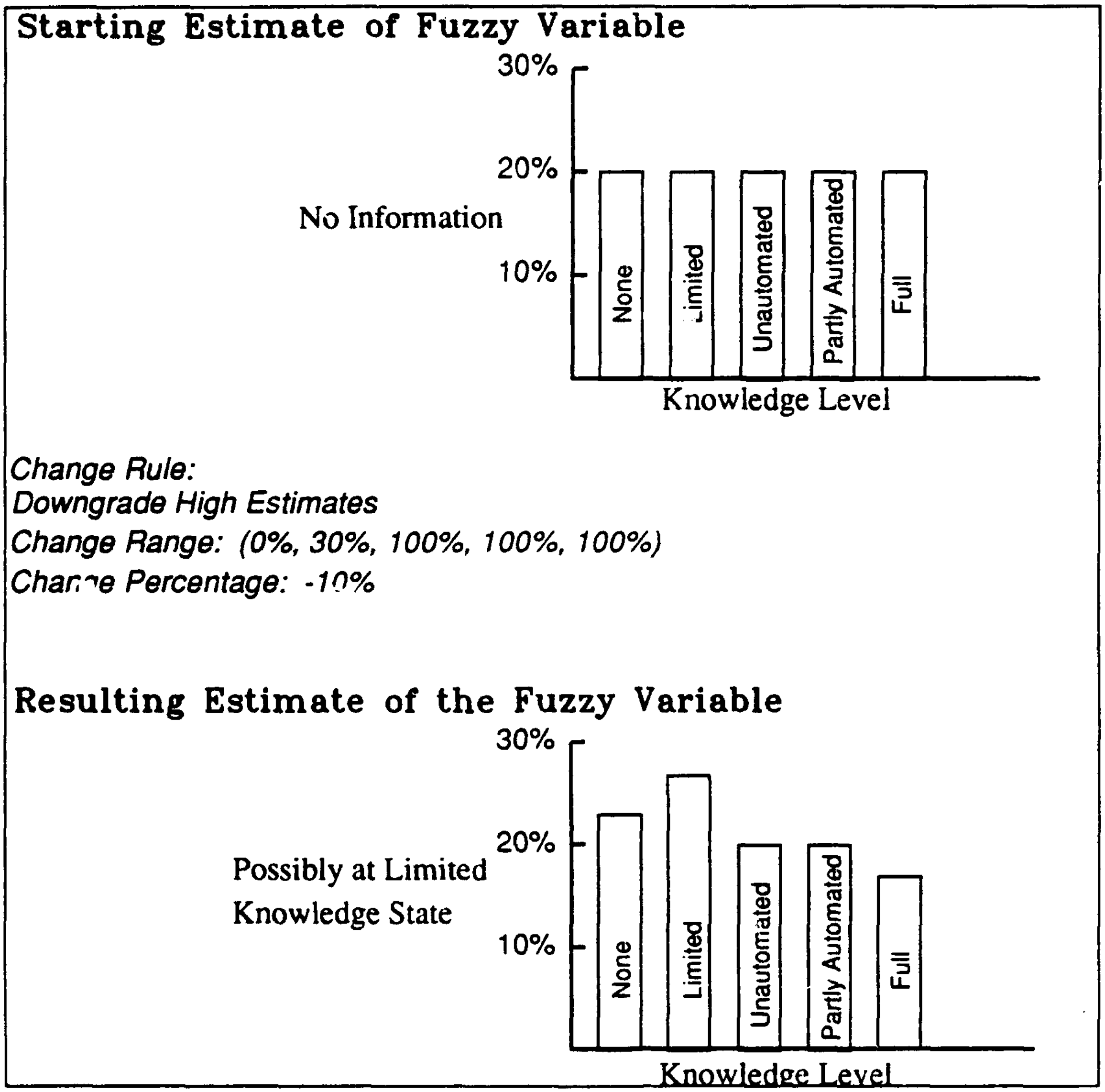


Figure 5. Fuzzy Variable Updating.

conceptual understanding, troubleshooting disposition, etc.) in determining the student's ability on the associated global knowledge component. Below are shown sample updating equations for global variables.

Test equipment usage = .40 strategic usage + .30 tactical usage + .30 probe placement skill

Space-splitting = .35 system understanding + .35 ability to interpret test results + .30 ability to use schematics

Attention to safety = .50 system understanding + .50 understanding of technical procedures

Summary. The main components of the student modeling knowledge base in Sherlock II are the student variable lattice, and the rules for updating the lattice's local and global variables. Each updating rule for a local variable consists of a performance condition, and an updating rate. The updating rate is controlled by a change percentage and change range. This catalogue of knowledge components suggests the following breakdown of tasks for building the knowledge base for a FSM component:

1. Identifying the local variables.

2. Identifying performance indicators (conditions) for updating local variables.

3. Assigning an updating rate to each performance condition, for each local variable that it is associated with.

4. Identifying aggregates of local variables, or global variables.

5. Assigning weights to variables in aggregate equations, which reflect the relative strength of each local variable as an indicator of the knowledge component described by the global variable.

As mentioned earlier, in building the prototype modeling component in Sherlock II, we relied primarily upon the results of cognitive task analysis to identify local variables, and expert ratings of student performance to identify global variables. We also relied upon our primary domain expert, Gary Eggan, to supply an initial set of performance conditions for local variables, updating rates, and weights in aggregation equations. These inputs are merely a starting point which must be tuned with experience in using Sherlock II. This summer's field testing will afford us the opportunity to adjust the knowledge base with respect to actual student data. More will be said about how we plan to make these adjustments below.

Updating an Individual Student Model

Acquiring information about the student. The expertise in object-oriented systems like Sherlock II is represented in computational "objects." An object is an independent piece of computer program that stores its own local data and can thus respond to various requests that other parts of the system might make of it. Knowledge about how to deal with specific components of the system being diagnosed is embedded in the objects that represent those components. For example, a component in a system will be represented by an object, and component objects will have routines relevant for the objects they represent. The object for a given component, such as a particular type of printed circuit card, might "know" how to draw the component it represents on the screen, how to model its component as part of an electronic circuit, how to test that component, and how to coach the trainee in his interactions with that component. It also will likely "know" how to record the information needed by the system to update its model of the student. Alternately, the component object can inherit some or all of this knowledge from its "parent" in the hierarchical network of objects that model the device (in Sherlock II, the test station), specializing the inherited routines if necessary. Thus, in object-oriented systems like Sherlock II, the device modeling needed for training and the student modeling needed for assessment and diagnosis are integrated via the system architecture.[16]

The information needed to model the student gets recorded by objects at two levels: the active circuit path, and individual component objects within this path. This information gets stored in an object that we call the *student trace*.[17] The circuit path object records "global" information such as the order of components tested, and the order in which circuit testing goals (e.g., verification that the measurement signal path is functioning ok) were achieved. Component objects record more "local" information about what happened while the student was testing a particular component—in particular, the order in which tests were made and component testing goals (e.g., verification that the inputs to the component are ok) were achieved; which particular pins were measured and how the probes were placed; what test equipment was used, and how it was set up, etc.

The updating procedure. Compared with the knowledge engineering requirements of the FSM approach, the tasks involved in updating an individual student model seem trivial. Some variables are updated dynamically, while the student is solving a problem. However, most variables are updated after problem completion, since updating takes time and would slow down the system if done completely dynamically. Dynamically updated variables mainly represent disruptive errors, such as probe reversals and incorrect settings on test equipment, which, if not corrected right away, could cause the student to flounder, with very little promise of instructional gain. Other dynamically updated variables correspond to safety hazards, such as attempting to conduct on ohms test when current is on. Dynamic updating is governed by rule "firings," as soon as these performance events occur. The current variable distribution is simply updated the amount specified by the updating rate.

Post problem-solving updating procedes as follows. First, Sherlock imports the student record, thereby getting access to the student's score on all local variables up to the current problem. If the student is new to the system, Sherlock initializes variables to a pre-specified level, as indicated by the "initial distribution" slot in Table I. Using its updating routines, Sherlock examines the student's solution trace, searching for performance conditions that trigger rule firings. Each indicator is represented in the *conditions table*. Sherlock simply records how many times each performance condition has been identified in the student trace. Each local variable associated with that condition will then be updated the recorded number of times. After each local variable has been updated, Sherlock uses its weighed equations to propagate these values "upwards" through the lattice, thereby updating global variables.

Software Engineering Tools Used to Develop the Prototype Modeling Component

[16] See Lesgold (in press) for a more in-depth discussion of how object-oriented programming supports the integration of training and assessment.

[17] Actually, the student trace consists of a *cluster* of objects, each one holding a different type of information—e.g., one records the student's measurements, one holds the sequence of achieved goals and the tests used to achieve these goals, another stores information about test device settings, etc. This division of labor makes updating the model more efficient, since individual updating routines need only access those objects that contain the information that they need. *However, for conceptual simplicity and ease of exposition, it is best to think in terms of one student trace* object.

Building a prototype system (or system module) presents the designers and developers of such prototypes with an interesting challenge. On the one hand, since the aim of the prototype is often to identify the tasks required to implement a given approach, and/or to assess the feasibility of carrying out the approach—a "proof of concept," so to speak—issues such as how well the prototype works and whether the underlying knowledge base is complete and correct are not central concerns. On the other hand, these issues can not be ignored entirely, because if the prototype does not work at all, it will be impossible to determine if it could be made to work. So, the system developer must find a happy balance between speed in building the prototype and quality.

We found it useful to build tools that would enable us to get the prototype FSM component in Sherlock II working as well as possible, until we could gather the data needed to fine-tune it. The primary development tool we built was what we call the **Student Model Browser**, which is essentially an interface for examining the student variable lattice and the current state of particular variables.[18] A snapshot of the browser in use is shown in Figure 6. By "clicking" on a particular variable node name (using a mousing device), the system developer can peruse subgraphs of the lattice in order to find out what the children of a particular node are (shown in the bottom half of the figure), and what local variables are associated with a selected global variable, thereby determining its value. These capabilities enabled us to visually detect some of the anomalies that result naturally from rapid prototyping of a hierarchical knowledge base, such as cycles and bi-directional dependencies.

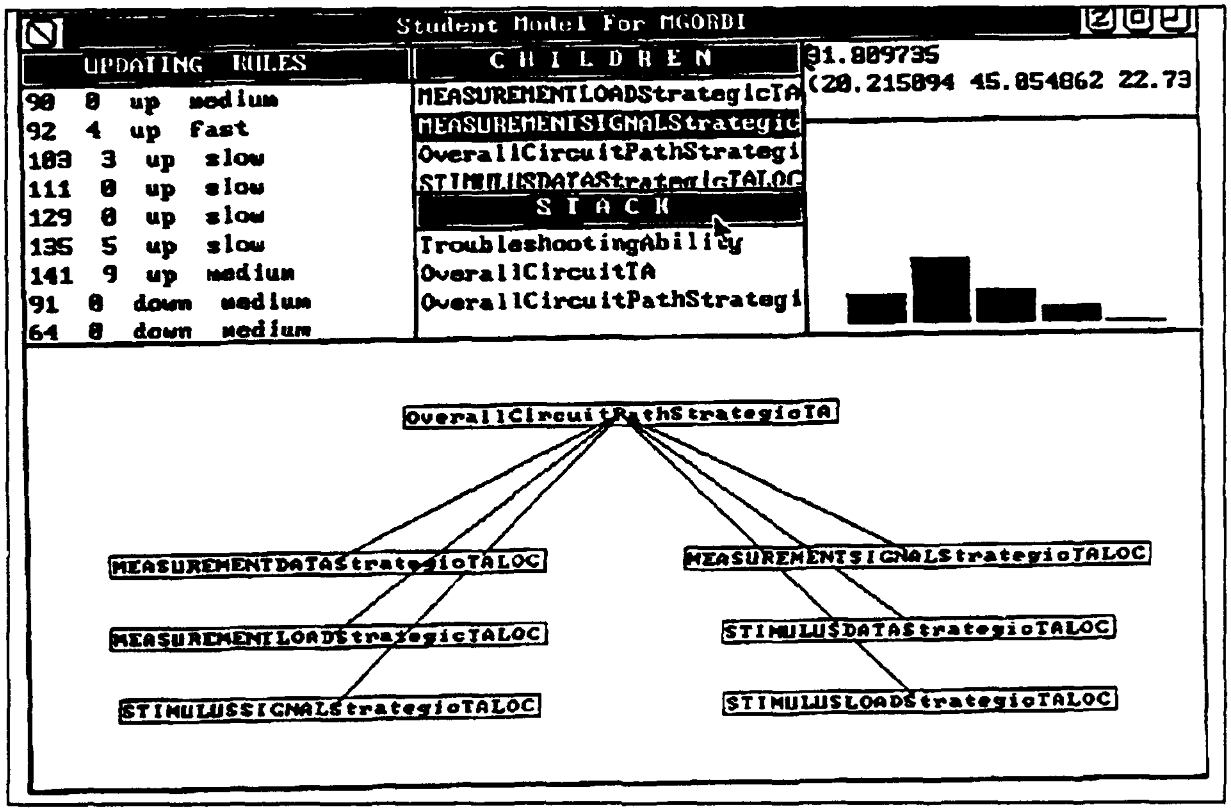
The **Browser** also displays the current distribution for a selected variable, and summarizes how that score was derived. Referring to Figure 6, the top right pane shows the student's current score on a local variable, *strategy for testing the measurement signal area of the test station*, with a histogram shown below the numeric distribution. The top left pane explains the score by displaying the updating rules that "fired,"[19] how many times they fired, the updating direction (upgrade or downgrade) and updating rate. Had the variable been global, its weighted equation would have been displayed.

These features of the **Browser** enabled us to determine if variables were being updated too slowly, or too quickly, for the problem-solving sessions of simulated experts and students that we ran on the tutor. Since updating rates often needed to be adjusted, we developed routines that would tell us what the revised change percentage should be.[20] Basically, these routines allowed us to specify how fast we would expect a very able student (or an expert) to reach expert level on a variable, in terms of the number of times we would expect a given updating rule to fire. The program would then tell us what the change percentage should be set to in order for the distribution to reach expert level in the specified number of firings.

[18] See Katz, Lesgold, and Gordin (in preparation) for a complete discussion of the **Student Modeling Browser**, and our plans to extend it into a tool that would be useful for trainers as well as system developers. Hawkes et al. (1990) present similar objectives in their discussion of a database system for accessing information about an individual student model in the TAPS tutor.

[19] The number identifies the performance condition that triggered the rule firing. The system developer is able to call up a window that will display the names of these conditions.

[20] We adjusted the change percentage, rather than the range, since the former is an easier structure to deal with and attempting to adjust both factors would have been way too complicated.



Student Model For MGORDI
UPDATING RULES
CHILDREN
STACK

```
90   0  up    medium
92   4  up    fast
103  3  up    slow
111  0  up    slow
129  0  up    slow
135  5  up    slow
141  9  up    medium
91   0  down  medium
64   0  down  medium
```

```
MEASUREMENTLOADStrategicTA
MEASUREMENTSIGNALStrategic
OverallCircuitPathStrategi
STIMULUSDATAStrategicTALOC
```

```
TroubleshootingAbility
OverallCircuitTA
OverallCircuitPathStrategi
```

```
01.809735
(20.215894 45.854862 22.73
```

OverallCircuitPathStrategicTA
MEASUREMENTDATAStrategicTALOC
MEASUREMENTLOADStrategicTALOC
STIMULUSSIGNALStrategicTALOC
MEASUREMENTSIGNALStrategicTALOC
STIMULUSDATAStrategicTALOC
STIMULUSLOADStrategicTALOC

Figure 6: The Student Model Browser

Needless to say, our estimates about how quickly a given variable should reach a certain level of expertise will often prove to be inaccurate, based on analysis of data gathered during field trials. We expect that identifying valid updating rates, and satisfying each of the other knowledge engineering tasks cited above will prove to be challenging. In the next section, we describe some of the techniques we might try to meet this challenge, in light of a proposed framework for evaluating a student modeling approach that will enable us to tell when we have done so.

PLANS TO IMPROVE SHERLOCK'S STUDENT MODELING COMPONENT

As several writers have pointed out (e.g., Woolf, 1988), surprisingly few tutoring systems have been evaluated, either as a whole, or to assess the effectiveness of particular modules.[21] In this section, we propose a framework for evaluating the student modeling component in an ITS. First, though, we wish to point out that there is an important difference between evaluating a *particular* implementation of an approach, such as fuzzy student modeling, and the approach per se. An evaluation of several systems needs to be conducted before conclusions can be drawn about the approach itself. With this in mind, we apply the proposed framework to our implementation of FSM, describe some of the hurdles that we will have to get over in order to meet the evaluation

[21] Littman and Soloway (1988) discuss these two aspects of evaluation for an intelligent tutoring system.

criteria, and present a repertoire of techniques we could try to do this.

Evaluating the Student Modeling Component in an ITS

There are three main dimensions to our proposed framework for evaluating a student modeling component: *validity, utility,* and *robustness.* The order mentioned corresponds to the order in which these dimensions should be considered in evaluating a student modeling component, since each criterion depends upon the one(s) that precede it. As the following overview will clarify, these criteria have their analogue in the three levels of instructional knowledge intrinsic to the curriculum theory underlying our modeling approach. (See discussion beginning on p. 6)

Validity. Validity signifies the extent to which the scores in the student models produced by the student modeling component are true—i.e., consistent with what an expert human rater would assign. In terms of Ohlsson's distinction between descriptive and pragmatic diagnosis referred to earlier (p. 6), validity focuses on the former. It measures how well the modeling component is able to carry out "research in miniature;" i.e., to identify which skills the student has, which he lacks.[22] Thus, validity focuses on the Knowledge Layer of the instructional network or—with reference to the particular modeling component in Sherlock II—on local student variables.

Utility. Utility represents pragmatic concerns: how well is the student modeling component serving the tutoring system? For example, how effective is Sherlock II's student modeling component in driving coaching and problem selection? Obviously, utility depends upon validity. The student modeling component will not serve didactic functions unless it is working properly. Since didactic decisions are generally made in response to curriculum goals, utility corresponds to the Curriculum Layer of the instructional model, which, in our system, includes the highest-level global modeling variables.

In addition to considering how useful a student modeling component is for a particular tutoring system, we can think of utility at a more global level: that is, how generalizable is the approach? To what extent could the student modeling component, or its parts (the knowledge base and updating procedures) be used in other systems?

Robustness. At the local level, robustness concerns the modeling component's ability to adapt itself to the idiosyncracies of an individual student; at the global level, its ability to accommodate differences between real-world and tutoring tasks. The more robust the student modeling component, the fewer cases will there be of students who score poorly on the system, but perform well on the job and vice versa. In order to be robust, the modeling component must first be working correctly, and able to satisfy didactic functions. But to be able to accurately model an unusual case, and to adapt itself to discrepancies between student performance on the system and on the job, the modeling component needs to be able to take into account the metaissues that comprise the third layer of Lesgold's model of instructional knowledge. Achieving this degree of adaptability is perhaps the greatest challenge facing the development of a technology for student modeling.

[22] Or, in "fuzzy" terms, avoiding the boolean distinction, which skills the student is strong in, which he is weak in.

In addition to the dependencies between the three evaluation criteria that we have mentioned, each criterion depends upon more specific evaluation factors. Validity depends upon the *accuracy* and *completeness* of components in the student modeling knowledge base. Utility depends upon *perspicuity*; student scores must be unambiguous. It also depends upon *efficiency*. Distributions that accurately reflect the student's level of ability have to be reached quickly, or the system will not be able to use these scores when it needs to. Robustness depends upon the modeling component's flexibility, or *adaptability*. Ultimately, the modeling component should be able to adjust its weights, change rates, etc. as it learns more about a particular student. In addition, it should be able to make similar adjustments based upon what it learns about discrepancies between students' performance on the system, and their performance on the job or on tasks that the system trained them to do.

Presently, we describe potential difficulties in meeting these criteria when implementing an FSM approach, as well as techniques we might try in order to fine-tune the prototype. For now, our main objective is to adjust the student modeling knowledge base, in order to increase validity. The other two dimensions (utility and robustness) can not be focused upon until the student modeling component is working properly, although the component's inadequacy in serving didactic functions (tailored coaching, problem selection, etc.) can point to sources of inaccuracy.

Achieving Validity

During field trials of Sherlock II, we will collect many samples of student problem-solving traces. We plan to have domain experts rate these traces, with respect to selected global (curriculum) variables. By comparing these expert ratings with the distributions produced by the student modeling component, we will be able to tell how well the modeling component is working. When we discover anomalies, we will have to trace the problem back to the values on the expert knowledge components, or local variables, and—further still—to the rates by which triggering events, or performance conditions, caused these variables to be updated. So, the limelight while fine-tuning for validity will be on local variables. Two main factors contribute to validity: the *completeness* and *accuracy* of information in the knowledge base.[23]

Completeness. The three main issues to address about the completeness of the knowledge base are:

1. Have all of the important expert knowledge components, corresponding to local variables, been identified?

2. Have all of the diagnostically significant behaviors in the domain, or performance conditions, been identified?

3. Have all of the performance conditions that map to each local variable been identified?

[23] Lesgold (in press) cites two other development objectives for the student variable lattice, *coherence* and *consistency*.

Identifying local variables. As we mentioned earlier, we did this mainly through cognitive task analysis. We expect, though, that analysis of student data will reveal the need for a more fine-grained analysis of the prerequisites for some of our local variables. For example, if students score poorly on *schematic-tracing ability*—as evident, for example, by frequent placement of probes on the wrong pins—we will want to decompose this knowledge component further, and represent the outcome as local variables beneath *schematic-tracing ability* in the variable lattice.

Identifying performance conditions. Cognitive task analysis has also been an important source of information about diagnostically significant behaviors. In addition, during policy-capturing sessions, our experimenters asked expert raters to point to specific behaviors that indicated the presence or absence of the skills that these raters deemed important in formulating their assessments.[24]

Certainly, we need to examine expert and student performance on the task that the tutor is trying to teach in order to identify behaviors that can act as indicators of student knowledge. But the target or *primary task* is not the only source of diagnostically significant behaviors. The system can also infer a lot about student knowledge by examining *student-system interactions* and students' behavior during *auxiliary tasks*—i.e., other instructional activities such as remedial exercises—that they can engage in, within the tutoring environment. We offer the following framework of performance situations as a heuristic for identifying diagnostically significant behaviors (expressable in updating rules as performance conditions) in a tutoring system.

1. **Primary task.** What things do system agents do while engaged in the primary task that indicates what they do or do not know? Most of the performance conditions in Sherlock II's student modeling knowledge base—e.g., reversed placement of probes, incorrect settings on testing devices, testing a component prematurely, failure to accomplish a component or circuit-level goal—pertain to the task of troubleshooting the test station.

2. **Student-system interactions.** As Gegg-Harrison (1990) and others (e.g., Brown & Ferrara, 1985) have pointed out, a teacher can learn a great deal about the depth of a student's understanding of a particular concept, or ability to perform a certain skill, by considering how much help the student needs in order to successfully complete tasks that exercise those concepts and skills. The more help the student needs, the less automated his or her grasp of the requisite knowledge components.

   In addition to considering the *amount* of help the student needs, human teachers also take into account the *nature of students' responses* to help while assessing students' ability. Elsewhere (Lesgold, Eggan et al., in press), we have proposed that tutoring systems should do the same. In coached practice environments such as Sherlock II, system help is graduated so as to ensure that students think on their own. The sooner in the help sequence that a student is able to succeed in achieving a problem-solving subgoal, the more knowledge components required for achieving that subgoal he must have control over. For example, if a student is able to carry out the next test required to verify that a component is working properly after receiving only two hints—one summarizing the tests

[24] Lesgold (in press) points out that some interference occurs when asking raters to explain their scores, since they might come up with reasons besides those that originally motivated them.

he has made so far, the other interpreting the results of these tests—the system should conclude that the student had difficulty interpreting his previous test results, and downgrade the corresponding modeling variable (*ability to interpret test results*). However, it should in addition upgrade the variables associated with all of the other concepts and skills needed to complete the test that the system did not have to "supply" to the student via coaching—e.g., knowledge about what pins to test (*schematic-tracing ability*), what type of test equipment to use and how to set it (*test equipment usage*), and which probe to place on each pin (*probe placement ability*).

3. **Auxiliary tasks.** In many tutoring systems, students can undertake other activities besides exercises that simulate the real-world problem-solving task that the tutor was developed to teach. There might be remedial exercises available on the system, for practicing skills that the student is weak in. For example, an electronics troubleshooting tutor such as Sherlock II could contain exercises on propagating voltage values through a circuit, or on tracing through schematics in order to identify the active circuitry for a particular diagnostic test. Students' behavior during these tasks could also be used to update the student model. Caution needs to be taken, though, in inferring that a student has grasped a concept or skill, given that he is able to succeed in carrying out these remedial tasks. The student still might lack understanding about when to *apply* this knowledge, in a real or simulated problem-solving situation.

In addition to remedial exercises, other tutor-based activities could serve as candidate sources of performance conditions. Students' behavior during collaborative tasks is one possiblity. As mentioned earlier, we plan to have students critique peer solution traces (real or simulated). We expect that the kinds of comments that students make will serve as an additional "window" through which to examine their knowledge. For instance, if a student critic fails to recognize an error or inappropriate action in the solution trace, or misdiagnoses an error, the system should infer that the critic also lacks some (or all) of the knowledge components associated with the target student's faulty action. Additional inference rules would be needed to select the most likely candidates from a set of potentially weak knowledge components.

Mapping performance conditions to local variables. The detailed cognitive task analysis that we conducted also revealed a lot about the associations between agent behaviors and the components of expert knowledge. During and following field trials, we plan to employ two additional techniques that will enable us to confirm and revise our condition-to-variable mappings. First, we intend to have experimenters—namely, experts in the task domain—interview students about their solution traces, using reflective follow-up mode to step through a given trace. In particular, experimenters will ask students to explain *why* they did or did not carry out certain actions—e.g., try to achieve a particular troubleshooting goal or carry out a particular test, at a particular time. We expect that students' responses will point to gaps in their understanding, thereby revealing links between student actions and the knowledge components represented by local variables. Second, we plan to use psychometric methods such as *factor analysis* and *cluster analysis* to identify potentially meaningful "clusters" of student behavior. Each cluster potentially defines a not-yet-identified knowledge component (Lesgold, Eggan et al., in press).

Accuracy. The main issue here is, do students' scores on particular variables, as recorded in fuzzy distributions, accurately reflect their skills? At the global variable level, accuracy depends primarily upon the *weights* attached to local variables in aggregation equations, and the accuracy of the local variable scores themselves. The latter, in turn, depends mainly upon the *rate* in which the current distributions are updated, each time a rule firing occurs.

Identifying correct weights. The updating rules for global variables (illustrated on p. 14) are, essentially, regression equations. We have hesitated to apply this term to the equations currently in the system, since they have not been derived using regression analysis. Field testing will produce the data needed to conduct regression analyses. The resulting regression coefficients will reflect the degree to which each subcomponent of knowledge (local variable) predicts the score on the dependent knowledge component (global variable).

To validate the weights derived from regression analysis, we plan to have experts specify the probability that an agent has a particular knowledge component, given that his score on the associated global variable is high—a technique drawn from Bayesian probability analysis.[25] The assigned probabilties should take into account both the likelihood of an expert possessing the skill, *and* of a novice having the same. As a case in point, it would be incorrect to claim that a local variable indicates expertise on a global variable 90% of the time, when even novices are 90% likely to have the skill represented by the local variable. Updating toward "expert" should occur if non-experts are only 50% likely to possess the skill, but it should be weaker updating than if non-experts are only 10% likely to have the skill.

Given strength of indicator probabilities such as these, we will also be able to run anomaly detection or "what if" experiments, using real or simulated traces of student and expert solutions. In effect, the experiments involve reasoning "backwards" from outcomes to causes.[26] For example, given a low score on a selected global variable, according to expert ratings of a student solution trace, we can see if the student modeling component also produced a low score on the variable. If not (i.e., the score came out significantly higher than the expert's rating), we should examine the scores on individual local variables. If we know that a particular local variable is a strong indicator for the global variable—based upon experts' probability assignments—and the system-generated score on that local variable is high, then we should examine the updating rules for the latter. Chances are good that the local variable is being updated too quickly, and the student is not as good at the local skill as the system is claiming.

Identifying correct updating rates. One way to do this has already been described. We can run the student modeling component on sample traces (of real or simulated experts or students), and just see if resulting distributions for local variables match those assigned by expert raters. If they do not, updating rates on rules that fired can be increased or decreased, as necessary. This is, essentially, what we have been doing in developing the prototype.

In addition, though, we could use similar probability theory-based techniques to those described

[25] The weights in our current set of aggregation equations, such as those shown on page 14, reflect a first-pass effort. We are talking here about a more rigorous assignment of probabilities. The ideas described here about using techniques derived from probability theory were suggested to us by our colleague, Robert Misleavy, at Educational Testing Service.

[26] As Misleavy (see footnote 25) points out, the experiments described in this paragraph are similar to techniques used to "train" connectionist networks.

above for fine-tuning the equations that update global variables. For example, experts could specify the probability that a particular behavior (i.e., performance condition) indicates the presence of a skill represented by a local variable. If the probability is high, and the student's score on a local variable comes out low despite the fact that the rule fired several times (i.e., the behavior manifested itself substantially), then the updating rate assigned to that performance condition should be increased.

Increasing Utility

Once we fine-tune the student modeling component to the point where it is producing scores that match our expectations, we can focus on how well it serves the system in performing didactic functions—e.g., tailoring advice to the student's level, and selecting problems. From this viewpoint, validity is not enough. Accurate scores have to be available when they are needed. They should reflect the student's ability on the curriculum variables that are relevant to didactic decision-making, and change as the students' ability changes. Thus, *efficiency* in reaching accurate scores is a major contributing factor for utility.

In addition, the student variable lattice should contain in its second layer (the Curriculum Layer) all of the global variables that will be needed for didactic purposes. Similarly, it is crucial to identify all of the local variables associated with each curriculum variable. Otherwise, the chance of the system making invalid inferences about a students' ability on high-level knowledge components increases dramatically (Kass, 1990). For example, the system might conclude that the student knows how to use test equipment because he can use a handheld meter and conduct wire tests (using the digital multimeter). But the student might not be able to use an oscilloscope. If the latter were absent from the variable lattice, and/or from the aggregation equation that determines the student's skill on test equipment usage, the system would make a false generalization about the student's ability to use test equipment. So, *completeness* is important for utility, as it is for validity.

To be usable, a fuzzy distribution has to be converted into a meaningful score. The system's ability to interpret a distribution, in order to assign an unambiguous score to it, is another major issue involved in achieving utility. We refer to this factor as *perspicuity*.

Finally, to be useful beyond the bounds of a particular system, the student modeling knowledge base should be generalizable, or at least contain a mixture of domain specific behavior about student ability and information that can be derived using general acquisition rules. We refer to this factor as *generalizability*.

Efficiency. One of the main obstacles to efficiency is that it takes time to gather enough data to be able to draw conclusions about how well a student is doing on particular variables. Human teachers face this problem, just as computer tutors do. We have two schemes in mind for dealing with this problem. First, we plan to compare the amount of change in the student's score on a particular variable with the same for a simulated expert, for all problems up to and including the current session. In other words, the amount of change in the expert's score would serve as a standard against which to interpret the student's score. If after, say, the first three problems the student reached a low "journeyman" level on schematic-reading ability, but a simulated expert only reached journeyman level after solving the same three problems, the system should conclude that the student is fairly good on the skill represented by this variable, since the difference between the student's and "expert's" scores is small. The system should then adjust the student's "raw" score accordingly. We have a great deal to learn yet about how to do this *differential scoring* (e.g., how to set thresholds for comparison), but the approach seems promising.[27]

In addition to differential scoring, we plan to analyze student traces in order to identify initial values for variable distributions that more closely reflect students' starting ability than the ones we set in the prototype.[28] We might find, for instance, that most students can use the handheld meter well, but not the oscilloscope. Thus, we would initialize the former to the high journeyman level, and the latter to the novice level. In other words, analysis of student problem-solving traces will enable us to "prime the scoring pump," so to speak.

One more method for determining student scores on variables in the face of insufficient evidence is suggested by the double-stereotype approach to modeling user knowledge implemented in KNOME, a UNIX advisory system (Chin, 1989). In KNOME, each fact about using the UNIX operating system is stereotyped according to its difficulty level (ie., *simple, mundane, complex,* or *esoteric*). In addition, users are stereotyped (as *novice, beginner, intermediate,* or *expert*) according to the difficulty level of the facts they know. The system relates user stereotypes with difficulty levels in terms of how many facts at a given difficulty level a user at a given ability level is likely to know. For example, experts should know all simple facts, and novices no esoteric (i.e., unlikely to be needed in using UNIX) facts.

Using this information, the system can carry out two types of reasoning. First, it can reason from the difficulty level of facts that a user has given evidence of knowing to the user's ability level. The system uses fuzzy (imprecise) terminology (eg., *likely, somewhat likely, unlikely*) to express its degree of confidence that a user belongs to a particular stereotype. Second, once the user has been stereotyped as a novice, beginner, etc., the system can use this stereotype to infer the likelihood that the user knows a particular fact, when not enough information has been accrued for the system to be able to assign a boolean value (true or false) to this relation. For example, the system can infer that a user does not know a particular complex fact, if the system believes that it is likely that the user is a beginner. Chin (1989) summarizes the reasoning ability afforded by double stereotyping as follows: "The double-stereotypes allow KNOME to combine a small number of explicit facts to first infer the user's level of expertise and from this, the user's knowledge of large sets of other facts." (p. 106)

In the current version of Sherlock, the system performs only the first type of reasoning. That is, it accrues evidence about student knowledge, in order to classify the student as a *novice, journeyman,* or *expert* troubleshooter. It does not, however, exploit the latter type of reasoning to draw inferences about the student's ability on scantily updated modeling variables. What would be required to enable the system to do this is to characterize each variable according to its

[27] Our colleague, Maria Gordin, suggested this approach.

[28] For the most part, we initialized local variables to the low journeyman level, as shown in Table I. Most trainees come to the tutor with some knowledge about troubleshooting, and enough experience not to be considered complete novices. In reality, this probably holds true for some variables, but not others.

difficulty level.[29] This, in turn, requires a model of cognitive growth within the domain, which captures how soon students tend to learn each knowledge component.[30] With the detailed task analyses that we have already performed, we are far along in being able to construct such a model. For example, we know that students can conduct ohms (resistance) tests before they can conduct voltage tests; can use the handheld meter before they can use the oscilloscope, etc. Additional analysis of student problem-solving traces should further contribute to our ability to assign difficulty ratings to knowledge components, which can, in turn, be used by the system to make inferences about the likelihood that the student is expert in a given knowledge component given what the system believes about his overall ability.

Completeness. As mentioned previously, our primary means of identifying global variables so far has been by applying policy-capturing techniques (Nichols et al., in press). We expect that the results of factor analysis will suggest additional global variables, as we discover novel clusters of local variables (Lesgold, Eggan, et al.). In addition, factor analysis will enable us to validate the associations that we claim hold between local variables, as expressed by our aggregation equations.

Perspicuity. We have identified three potential sources of ambiguity in interpreting fuzzy distributions. First, there is lack of clarity about the meaning of the central interval (e.g., the third interval, in a 5-interval distribution). In particular, does a distribution whose peak is located at or near the center mean that the student is mediocre on the skill, or that the system can not decide if the student is or is not in the set of experts for that skill? As Derry (personal communication) pointed out to us, the difference does not matter, as long as the system designers define the meaning a priori, and make the system behave consistently.[31]

The second potential source of ambiguity involves loss of information when the system converts distributions to scalar values. This problem is illustrated in Figure 7, which shows three distributions that convert to a score of "3."[32] Obviously, more is known about the top right and bottom distributions, than the top left distribution. One possible way around this problem is to keep the distribution around after it has been converted, and define rules that "intelligently" interpret converted scores in reference to its corresponding distribution.

Finally, there is the problem of interpreting a distribution that matches the value to which the associated local variable has been initialized. How is the system to "know" if this distribution is in its initial state, or has *returned* to this state, after having been updated a few times? More is known about the latter, since it was derived from recent evidence about a particular student's

[29] Sterotyping according to difficulty level is better suited to conceptual and skill variables, than to dispositional variables, such as *tendency to test components, rather than replace them*. For the latter, we could instead categorize according to immediacy of acquisition, in the course of trainee development. For example, *tendency to test components, rather than replace them* is uncharacteristic of many novices, who tend to swap components rather than test them. So, we would assign a "middle" rating to the variable, since it neither happens early or late in a trainee's development.

[30] McCalla & Greer (1990) also discuss the value of genetic models of learning within a domain for student modeling.

[31] In our system, we assign the first interpretation described here to the central interval; Hawkes and her colleagues assign the latter in the TAPS tutor (Hawkes et al., 1990).

[32] For example, the top left distribution can be converted as follows: .2(1) + .2(2) + .2(3) + .2(4) + .2(5) = 1/5 + 2/5 + 3/5 + 4/5 + 5/5 = 15/5 = 3.

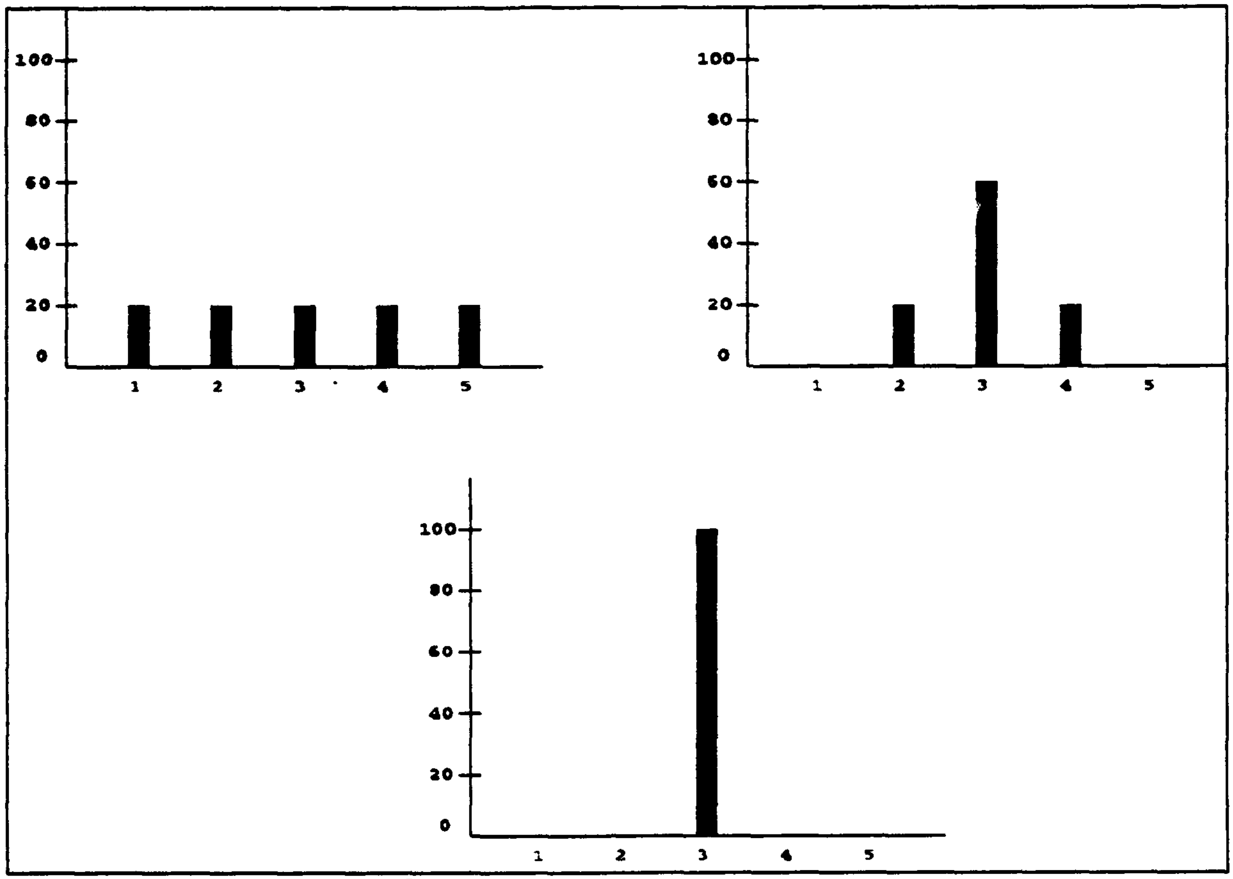


Figure 7: Distributions that Convert to 3

behaviors, as opposed to an expert's estimate of general student ability. So, it seems desirable to heed this distinction. One possiblility is to flag a variable once it has been updated.

Generalizability. One of the weaknesses of our approach to building the student modeling component in Sherlock II is that we specified, by hand, each updating rule, for each local variable. In retrospect, we see that we probably did not have to. The conditional part of several updating rules—i.e., the performance conditions that "trigger" rule firings—could have been generated automatically using general rules for acquiring information about an agent, such as those described in Kass (1990). We would then have needed only to specify an appropriate updating rate for each automatically specified condition.[33] As we discuss below, we expect to run into some difficulties doing this. But if after experimenting with this idea we decide to incorporate general acquisition rules in future versions of the system, it will be much easier to extend the knowledge base for use in related task domains—e.g., troubleshooting a different kind of test station.

[33] In fact, we expect that it would be useful to extend Kass's rule set by focusing on behavioral acts, as opposed to speech acts, from which Kass's rule set has primarily been derived.

We can illustrate what we have in mind with reference to one of Kass's rules. Many of the updating rules that we stated for local variables are instantiations of what Kass calls the Sufficiency Rule, which "is based on the maxim of quantity, which states 'make your contribution as informative as necessary, but not more so.' If the user is cooperating, this means he or she should do all that is necessary to help the system achieve its goal." (p. 22) In Sherlock II's device model, every component object "knows" what actions are required to achieve each of its goals. According to the Sufficiency Rule, then, failure to carry out a necessary action indicates either that the student is weak in the skills needed to perform the action, or has insufficient understanding about the goal itself—in particular, about what actions are required to achieve it.

Given this knowledge about component-testing goals, and the Sufficiency Rule, the system could specify updating rules for component-testing variables and for the skills needed to carry out each testing action. For example, the system could generate a rule that states that the local variable for knowing how to test logic cards should be downgraded (an amount to be specified based on expert input), if a student fails to verify the outputs of a logic card. An additional rule would state that the local variable for *ability to carry out voltage tests* should be downgraded (again, by a specified amount) under the same circumstances (i.e., the student fails to test the outputs of a logic card), since this ability is also necessary for satisfying the component-testing goal.

Of course, some caution needs to be taken, lest the system generate too many updating rules, many of which are invalid. Referring to the above example, the system could identify *all* of the prerequisite skills for testing the outputs of a card, and all of the prerequisites for these skills, etc., and recursively generate rules for downgrading all of these skills if the triggering event (failure to test outputs) occurs. For example, the system could generate a rule for downgrading probe placement, since this is one of the skills required for testing the outputs of a logic card. However, a causal connection between inability to place probes correctly on pins and not testing outputs is too remote a possibility to validate specifying such a rule. Consequently, some filtering of automatically generated rules, or more stringent constraints on rule generation in the first place, must occur in order to use general acquisition rules effectively.

Achieving Robustness

Student modeling components are typically built with the "general student" in mind. Similarly, they are aimed at modeling students' performance on exercises that simulate real-world tasks. Some amount of artifice is inherent in each of these attempts. An individual student's behavior, learning style, and progress could be orthogonal to that of the "general student." For example, McCalla & Greer (1990) cite research showing that students sometimes undergo "revolutionary conceptual shifts" (p. 6) in their mental models of recursion, in contrast to the gradual conceptual changes that the general student is expected to undergo. And, as we have mentioned earlier, success on academic exercises is no guarantee of success on real-world tasks, just as the latter does not necessarily demand understanding of all aspects of the subject domain. So the system should be able to adjust to discrepancies between students' performance on the system, and on the job.

Elsewhere (Lesgold, Eggan, et al, in press) we have described some techniques for enabling the system's modeling component to adapt itself in response to idiosyncracies of the individual student, and to discrepancies in students' task performance on and off the system. The essential idea stems from connectionist learning theory, and involves specifying events that will trigger the modeling network to adjust its updating weights, rates, etc., as it is run on training samples—i.e., real or simulated traces of expert and student performance. For example, one such "triggering event" might be that a student fails after receiving a hint that the system predicted would work, based upon the student's current level of ability. In response to this failed expectation, the system should decrease the updating rates for variables used by the system to assess the student's ability level for coaching purposes.

Table III summarizes the discussion presented in this section. It displays each criterion for assessing a student modeling component in our proposed evaluation framework, with knowledge engineering techniques that could be used to meet these criteria. We remind the reader, though, that we have not yet tried most of these techniques.

Table III: Criteria for Evaluating a Student Modeling Component and Associated Knowledge Engineering Techniques

Evaluation Criterion	**Contributing Factors**	**Knowledge Engineering Techniques**
Validity	**Completeness**	**Local variables**: Cognitive task analysis
		Performance conditions: (1) Cognitive task analysis (2) Asking raters to explain scores in terms of student behaviors (3) Identifying diagnostically significant behaviors in the *primary task, student-system interactions*, and *auxiliary tasks.*
		Mappings of performance conditions to variables: (1) Cognitive task analysis (2) Asking students to explain their actions
	Accuracy	**Weights in aggregation equations**: (1) Regression analysis (2) Expert assignment of probability that student has local knowledge component given that he has global component (3) Adjust by running "what if" experiments on sample solution traces
		Updating Rates: (1) Comparison of system scores with experts' scores (2) Expert assignment of probability that student performed a particular action, given that his score on the associated local variables is high
Utility	**Efficiency**	(1) *Differential scoring:* Comparison of degree of change in student's score, with that in expert's score (2) Initializing distributions to more closely represent "general student" (3) Stereotype student according to overall ability, then use stereotype to infer likelihood that student knows particular concepts, skills, etc.
	Completeness	**Global variables**: (1) Policy-capturing techniques (2) Factor analysis
	Perspicuity	(1) Assign unambiguous meaning to distribution intervals (2) Retain distribution offer it has been converted to a scalar value (3) Flag variables once they have been updated
	Generalizability	Use general acquisition rules
Robustness	**Adaptability**	Use sample traces to "train" student variable lattice; specify "triggering events" for system self-modification—in particular, for revising updating rates

CONCLUSIONS

Although it is much too early to assess fuzzy student modeling as an approach, our experience with developing a prototype FSM component for Sherlock II, coupled with Hawkes et al.'s (1990) use of FSM in TAPs, suggests that the approach is promising, tractable, and extendable. There is no question that the knowledge engineering requirements are difficult and demanding. But we believe that we will be able to meet these requirements through some of the means we have described (e.g., simulation studies, statistical methods, interviews with students and trainers) and by building development tools such as the Student Modeling Browser. We now need to try out some of the techniques for tuning the student modeling knowledge base that we have described, in order to determine how tractable the task of building an FSM component that is not only valid and useful, but also generalizable and robust, really is.

We also believe that the FSM approach can be extended to handle more detailed levels of diagnosis, should research on how teachers model students reveal that teachers do, in fact, at times construct precise models of students' understanding and misconceptions. As Derry has recently demonstrated, the approach is also extendable to modeling metacognitive processes (Derry, in press). To this list, we would add that FSM techniques need not be restricted to intelligent tutoring systems. As Chin (1989) has already shown, fuzzy modeling techniques can also be incorporated into the user modeling component of natural-language dialogue systems.

But the main lessons we have learned from our experiment with FSM have nothing to do with the approach per se; rather, they pertain more to developing intelligent tutoring systems in general. The belief that theory and empirical research should inform instructional system design has essentially become an aphorism for both classroom teaching and ITS development. Our experience with implementing the FSM approach has made this principle more vivid to us. As we have tried to demonstrate, we discovered that empirically-grounded theories about curriculum design, system agents and agent modeling can not only inform the development of an ITS, but its evaluation as well. In turn, applying an evaluation framework such as the one we have proposed can further inform system development, by identifying limitations to tried approaches, and by prompting the identification of techniques for overcoming such limitations.

Another principle ingrained within the literature on developing an ITS technology is that the big problems have to be addressed in an interdisciplinary manner (e.g., McCalla & Greer, 1990). Again, our experiment with FSM supports this belief. For example, some of the fine-tuning techniques that we will try stem from Bayesian probability theory; others are rooted in connectionist learning theory. The idea of incorporating general acquisition rules within the knowledge base stems from the field of user modeling, particularly in natural language dialogue systems (Kass 1990), as does the idea of reasoning from stereotypes about users' ability to their ability on particular knowledge components (Chin, 1989). Kass (1989) has argued that those involved with the enterprise of building user modeling components for dialogue-driven systems should heed the lessons to be learned from the work done on student modeling. We believe that the reverse should happen too.

REFERENCES

Anderson, J. S., Boyle, C. F., & Reiser, B. J. (1985). Intelligent tutoring systems. *Science, 228,* pp. 456-468.

Brown, A., & Ferrara, R. (1985). Diagnosing zones of proximal development. In J. Wertsch Ed.), *Culture, Communication, and Cognition: Vygotskian Perspectives.* Cambridge: Cambridge University Press, pp. 273-305.

Chi, M.T.H., & Bjork (in press). *Modeling expertise.* Prepared for the National Research Council. National Academy Press.

Chin, D.N. (1989). KNOME: Modeling what the user knows in UC. In A. Kobsa & W. Wahlster (Eds.), *User models in dialog systems* (pp. 74-107). New York: Springer-Verlag.

Cohen, P.R., Perrault, C.R., & Allen, J.F. (1981). Beyond question answering. In Lehnert & Ringle (Eds.). *Strategies for Natural Language Processing.* Lawrence Erlbaum: NJ, pp. 245-74.

Cohen, R., and Jones, M. (1989). Incorporating user models into expert systems for educational diagnosis. In A. Kobsa & W. Wahlster (Eds.), *User models in dialog systems* (pp. 313-333). New York: Springer-Verlag.

Derry, S. (in press). Metacognitive models of learning and instructional systems design. In P.H. Winne & M. Jones (Eds.), *Foundations and Frontiers in Instructional Computing Systems.*

Elsom-Cook, M. (1989). Guided discovery tutoring, preprint for Nato Workshop on Guided Discovery Tutoring, Italy.

Fox, B. A. (1988a). *Repair as a factor in interface design.* Technical Report. Boulder: Institute for Cognitive Science, University of Colorado.

Fox, B. A. (1988b). *Robust learning environments: The issue of canned text.* Technical Report. Boulder: Institute for Cognitive Science, University of Colorado.

Gegg-Harrison, T.S. (1990). Zoning in on student models: Modeling cognitive potential in a schema-based Prolog tutor. *Proceedings of the 2nd International Workshop on User Modeling,* March, 1990. Honolulu, Hawaii.

Ginsburg, H.P. (1983). Cognitive diagnosis of children's arithmetic. *Issues in Cognition: Proceedings of a Joint Conference in Psychology,* 287-300. Washington, DC: National Academy of Sciences and American Psychological Association.

Hawkes, L. W., Derry, S.J., & Kandel, A. (in press). Fuzzy expert systems for an intelligent computer-based tutor. In A. Kandel (Ed.), *Expert Systems in the fuzzy age.* Reading, MA: Addison-Wesley.

Hawkes, L.W., Derry, S.J., & Rundensteiner, E.A. (1990). Individualized tutoring using an intelligent fuzzy temporal relational database. *International Journal of Man-Machine Studies, 33,* 409-429.

Kass, R. (1990). Building a user model implicitly from a cooperative advisory dialog. *Proceedings of the 2nd International Workshop on User Modeling,* March, 1990. Honolulu, Hawaii.

Kass, R. (1989). Student modeling in intelligent tutoring systems—Implications for user modeling. In A. Kobsa & W. Wahlster (Eds.), *User models in dialog systems* (pp. 386-410). New York: Springer-Verlag.

Katz, S., Lesgold, A., & Gordin, M. (in preparation). Preliminary design of a trainer interface. *LRDC Technical Report.*

Katz, S., & Lesgold, A. (in press). The role of the tutor in computer-based collaborative learning situations. In S.P. Lajoie & S. Derry (Eds.), *Computers as Cognitive Tools*, NJ: Lawrence Erlbaum Associates.

Lajoie, S. P., & Derry, S. (in press). Introduction: Computers as cognitive tools. In S.P. Lajoie & S. Derry (Eds.), *Computers as Cognitive Tools*. NJ: Lawrence Erlbaum Associates.

Lajoie, S.P., & Lesgold, A. (1989). Apprenticeship Training in the Workplace: Computer-Coached Practice Environment as a New Form of Apprenticeship. *Machine-Mediated Learning, 3,* 7-28.

Lepper, M.R. (1989). Goals and strategies of expert human tutors: cognitive and affective considerations. Paper presented at the *Annual Meeting of the American Educational Research Association*, San Francisco, CA, March 1989.

Lepper, M. R., Aspinwall, L., Mumme, D., & Chabay, R. W. (in press). Self-perception and social perception processes in tutoring. Subtle social control strategies of expert tutors. In J. Olson & M. P. Zanna (Eds.), *Self inference processes: The sixth Ontario symposium in social psychology.* Hillsdale, NJ: Erlbaum.

Lesgold, A. (in press). Assessment of intelligent training systems: Sherlock as an example. In E. Baker and H. O'Neil, Jr. (Eds.), *Technology assessment: Estimating the future.* (Tentative title). Hillsdale, NJ: Erlbaum.

Lesgold, A. (1988). Toward a theory of curriculum for use in designing intelligent instructional systems. In H. Mandl & A. Lesgold (Eds.), *Learning Issues for Intelligent Tutoring Systems.* NY: Springer-Verlag.

Lesgold, A.M., Eggan, G., Katz, S., & Rao, G. *(in press).* Possibilities for Assessment Using Computer-Based Apprenticeship Environments. To appear in W. Regian & V. Shute (Eds.), *Cognitive approaches to automated instruction.* Hillsdale, NJ: Erlbaum.

Lesgold, A.M., Lajoie, S.P., Bunzo, M. & Eggan, G. (in press). Sherlock: A coached practice environment for an electronics troubleshooting job. In J. Larkin, R. Chabay, & C. Scheftic (Eds.), *Computer assisted instruction and intelligent tutoring systems: Establishing communication and collaboration.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Lesgold, A. (1988). Toward a theory of curriculum for use in designing intelligent instructional systems. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligent tutoring systems* (pp. 114-137). New York, NY: Springer-Verlag.

Littman, D. & Soloway, E. (1988). Evaluating ITSs: The cognitive science perspective. In M.C. Polson and J.J. Richardson (Eds.), *Foundations of intelligent tutoring systems* (pp. 209-242). Hillsdale, NJ: Lawrence Erlbaum Associates.

McCalla, G. and Greer, J. (1990). Tracking student knowledge. *Proceedings of the 2nd International Workshop on User Modeling,* March, 1990. Honolulu, Hawaii.

McCalla, G., Greer, J.E., and the SCENT Research Team (1988). Intelligent advising in problem solving domains: the SCENT-3 architecture. *Proceedings of the International Conference on Intelligent Tutoring Systems (ITS '88)*, Montreal, June, pp. 124-31.

McArthur, D., Stasz, C., & Zmuidzinas, M. (1990). Tutoring techniques in algebra. *Cognition and Instruction, 7*(3), pp. 197-244.

Moore, J.D. (1989). A reactive approach to explanation in expert and advice-giving systems. *PhD thesis, Univerisity of California, Los Angeles.*

Nichols, P., Pokorny, R., Jones, G., Gott, S.P., & Alley, W.E. (in press). *Evaluation of an avionics troubleshooting tutoring system.* Special Report. Brooks AFB, TX: Air Force Human Resources Laboratory.

Ohlsson, S. (1987). Some principles of intelligent tutoring. In R.W. Lawler and M. Yazdani (Eds.), *Artificial intelligence and education (Volume 1): Learning environments and tutoring systems* (pp.203-237). Norwood, NJ: Ablex Publishing.

Putnam, R.T. (1987). Structuring and adjusting content for students: A study of live and simulated tutoring of addition. *American Educational Research Journal, 24*(1), 13-48.

Reiser, B.J. (1989). Pedagogical strategies for human and computer tutoring (Tech. Rep. No. CSL Report 38). Princeton, NJ: Congitive Science Laboratory, Princeton University.

Soloway, E. (1990). Interactive learning environments. Paper presented at the NATO Advanced Studies Institute, Calgary, July.

Sparck Jones, K. (1989). Realism about user modeling. In A. Kobsa & W. Wahlster (Eds.), *User models in dialog systems* (pp. 386-410). New York: Springer-Verlag.

Woolf, B., & McDonald, D. (1984). Building a computer tutor: design issues. *Computer 17*, 61-73.

Woolf, B. (1988). Intelligent tutoring systems: a survey. *In Exploring Artificial Intelligence*, papers from the 6th and 7th National Conferences on AI, Philadelphia and Seattle.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control, 8.*

Graphics Tools in Smalltalk/V
by
The University of Pittsburgh,
Learning Research and Development Center

This document was prepared by Edward F. Hughes, at and for the University of Pittsburgh, LRDC.

Table of Contents

Figures

Text Boxes

Abstract

Many high-level languages and library packages provide useful mechanisms and routines that allow the application programmer to focus on the behavior of the application instead of the implementation of windows. Smalltalk, for instance, provides the Model-View-Controller paradigm (henceforth referred to as MVC). These mechanisms give a standard way to quickly build window applications.

In the same way, the University of Pittsburgh's Learning Research and Development Center (LRDC) has created a graphics package that provides a systematic way to easily add graphic displays and interactive graphics and video to a window application. The LRDCGP's approach encourages the development of reusable objects whose behavior is kept separate from the mechanics of displaying their graphic representation. The approach is entirely consistent with the object oriented paradigm and the underlying windowing system. This consistency allows an application programmer who is already adept with building windowed applications on the chosen platform to easily incorporate graphics.

The LRDC's Graphics Package (LRDCGP) is implemented in Digitalk's Smalltalk V/286 (henceforth referred to as Smalltalk). This implementation of Smalltalk uses the Model-View-Controller paradigm, and the LRDCGP is designed to augment this windowing implementation. The LRDC is currently investigating a port to Digitalk's Smalltalk for Microsoft Windows, which does not use the MVC paradigm. Due to the encapsulation of the participating classes in the LRDCGP, porting to the new platform is expected to be relatively easy.

1 Windows Programming in Smalltalk V/286

Since the LRDCGP has been implemented as an augmentation of the Smalltalk's windowing classes, these classes will be briefly discussed here. Emphasis will be given on the motivations behind Smalltalk's windowing classes that are shared by the LRDCGP.

1.1 A Quick Object Oriented Decomposition of Windowing Systems

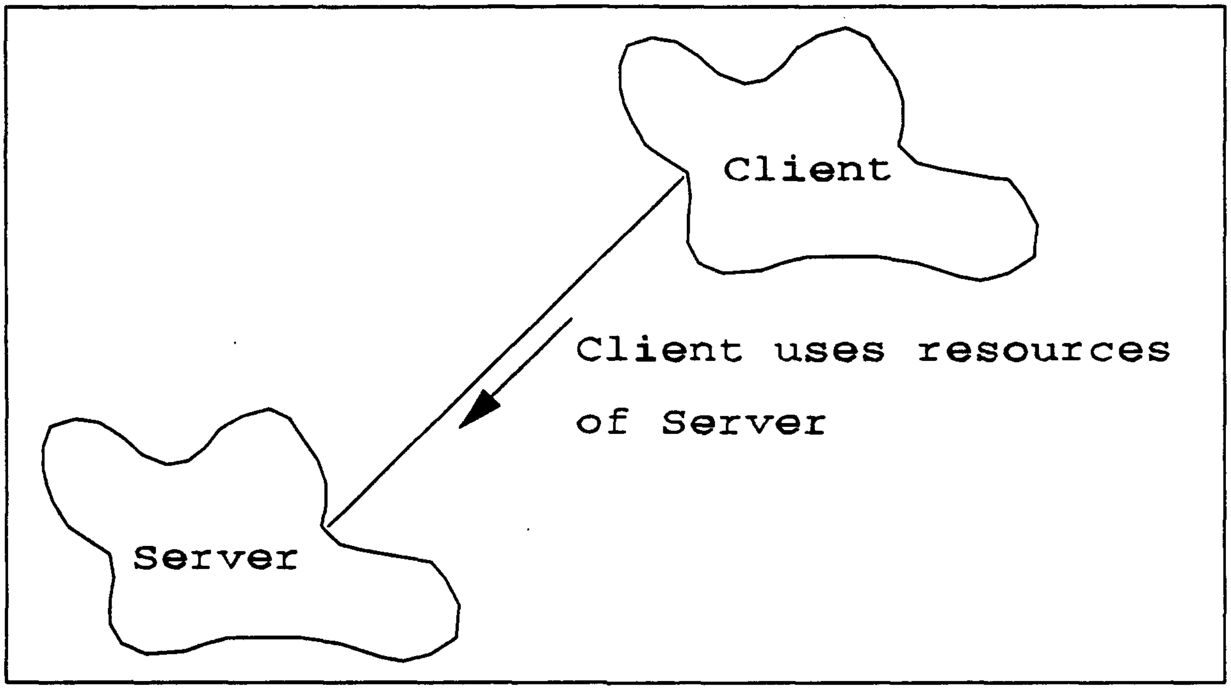


Figure 1 Client / Server Relationship

One very important part of object oriented analysis and design is the identification and isolation of the participants of a task. These participants are implemented as instances of classes. Classes encapsulate the behavior of their instances. Much of the internal workings of objects is known only to the object itself. Participation of that object in a task is requested by sending messages to the object.

In a windowing system, you might identify the object that you want to display graphically, the object that you want make the display, and an object which will capture user interaction and to send appropriate messages to display object.

There are different degrees to which objects interact with each other. Some objects use others, either by referring to their states, or by operating on them. Such objects are called clients (see Figure 1). Other objects are only operated on by other objects, i.e. clients, but never operate on other objects themselves. These objects are therefore not clients. Instead, they are servers, serving the needs of other objects.

It is very important to realize that it is preferable to keep the display mechanism separate from the application using this mechanism. This makes it both easy to implement applications that use windows as well as easy to port the application to other systems that use other display mechanisms. Therefore, the application is a server of the windowing mechanism. All user interactions are to be intercepted by the windowing mechanism, and this mechanism will inform the application of changes made, and make requests for new information to be displayed. The application may never directly send messages to the windowing mechanism. It may only update itself. The bridge between the application's request to update itself and the windowing mechanism which displays the update must be supplied by the programming platform used. In Smalltalk, such updates are provided by the Dependency mechanism.

1.2 Model-View-Controller Paradigm

We have noted how separating the windowing mechanism from the application simplifies application programming and porting to other systems. This section will discuss the components used in the MVC windowing mechanism, and how they interact with the application.

1.2.1 Model: The Object to be Displayed

The separation of windowing implementation and application means that the programmer need only be concerned with the objects to be displayed in each pane given its current state, and how this state should be changed when informed of user interaction.

A window application may have one or many software entities that it chooses to display. Each entity knows its state, how to respond to state changes made by user interactions, and how to give an appropriate representation of itself for display when asked for one. These entities are known as software *models*, since they model some abstraction.

Of course, the application itself may be a software model. This model's states would include its other models and all other information to be displayed. Very often, this is the only model that can be identified in an application. For instance, in Smalltalk's class hierarchy browser, there are several display areas that enable the user to browse through the classes and their methods. The application class, ClassHierarchyBrowser, models the entire behavior of the application. This is because no part of it could be isolated as a reusable software model.

Given a particular change in one of the model's panes, the model may decide that the contents of another pane should be updated. This is done via the Dependency mechanism. The model tells itself that a particular pane, named, say, *subpane*, is to be updated. Updates are implemented by broadcasting the contents message of the pane to all of the dependents of the model. In Smalltalk/V 286, the contents message of a pane is its name. Since the model's dependents include all of the panes, when the pane of name *subpane* receives the broadcasted message, it knows to update itself. An example of this is illustrated later in text box 4.

1.2.2 View: The Window and Panes

The view is the embodied in the window and the pane. The window, in Smalltalk/V 286 is an instance of TopPane. TopPane takes care of mundane book-keeping, such as the window's position, its size, and its color.

Information that is commonly displayed may be shown is several forms: graphics, text, and lists. Smalltalk provides corresponding views.

1.2.2.1 GraphPane

Instances of GraphPane are used to display bitmaps. Its contents method must return either a~ instance of Form, which contains a bitmap, or nil. If the contents method returns nil, then the GraphPane copies the display area encompassed by its frame.

1.2.2.2 TextPane

TextPanes are used to display text. Its contents method must return an instance of String.

The TextPane (via the text editing capabilities of its dispatcher, to be described later) allows the user to temporarily change the contents of the pane. But the model is not informed of such changes until the user chooses to save them. Once the user has made this decision, the model is informed of its new state by sending the model the change selector for the text pane along with the text contents being displayed.

1.2.2.3 ListPane

ListPanes are used to allow the user to select from a list of strings, ordered vertically. The contents method must return a collection of instances of String.

When the user selects one of the text items in the list, the model is informed of this selection by sending it the pane's change selector with a reference to the selected item. The reference may be either the contents of the list item selected, or the numeric position of the item in relation to the list, depending on how the pane was initialize.

1.2.3 Controller: The Dispatcher

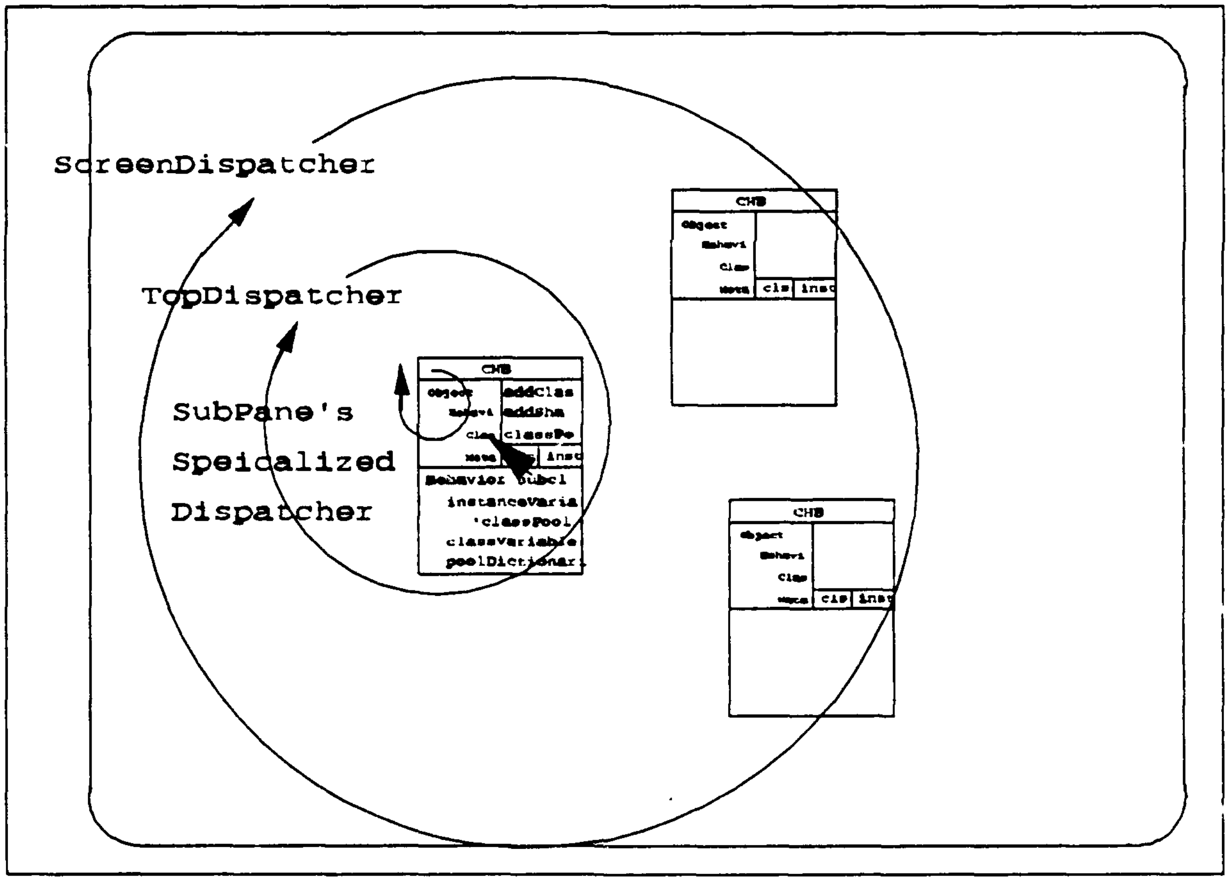


Figure 2 Smalltalk's Dispatchers

The controller responds to user interactions. In Smalltalk/V 286, controllers are called Dispatchers. The dispatchers are used to capture user events, and to delegate processing.

Figure 2 indicates the different levels of controllers. At the top is the Scheduler, the sole instance of DispatchManager. Its job is to trap user interaction, find the window the user interacted with, and to inform the dispatcher for that window that it is receiving user interaction.

The dispatcher for a window is an instance of TopDispatcher. The TopDispatcher for the selected window then finds the subpane in which the user is interacting. It is the subpane that actually reports the user's interaction to the model.

1.2.3.1 GraphDispatcher

When a user clicks over an instance of GraphPane, its model is informed of this change in its state by sending the model the change selector for the pane along with the location of the mouse in the pane in terms of the pixel offset from the pane's origin.

1.2.3.2 TextEditor

As its name would suggest, instances of TextEditor have text editing capabilities. The TextEditor allows the user to temporarily change the contents of the pane. But the model is not informed of such changes until the user chooses to save them. Once the user has made this decision, the model is informed of its new state by sending the model the change selector for the text pane along with the text contents being displayed.

1.2.2.3 ListPane

When the user selects one of the text items in the list, the model is informed of this selection by sending it the pane's change selector with a reference to the selected item. The reference may be either the contents of the list item selected, or the numeric position of the item in relation to the list, depending on how the pane was initialize.

1.2.3 View and Controller as Components of One Abstraction

Figure 3 shows how the view and controller interact with each other and with the model.

The link between the view and the model shows how the model is a server to the view. When the view is informed that it is to update itself, it must ask the model for the relevant information that it is to display.

The link between the controller and the model shows how the model is a server to the controller. When the controller is informed of a user's input to the pane that it controls, it must inform the model of its new state change.

It has been shown that each view has a specialized controller. They are paired in a meaningful fashion. The controller mirrors the user's intent of his or her interaction with the displayed information.

For instance, if a user were to choose an item from a list displayed in an instance of ListPane, it should not matter where exactly over that list item the user clicked. The model is less interested in the exact pixel over which the user clicked than the item the user intended to select. Thus, a GraphDispatcher would be of less use in conjunction with a ListPane than a ListEditor. In this way, the two classes combine to model a particular type of interactive display. The application programmer is abstracted away from *how* a list item was selected, and instead may focus on *what* item was selected.

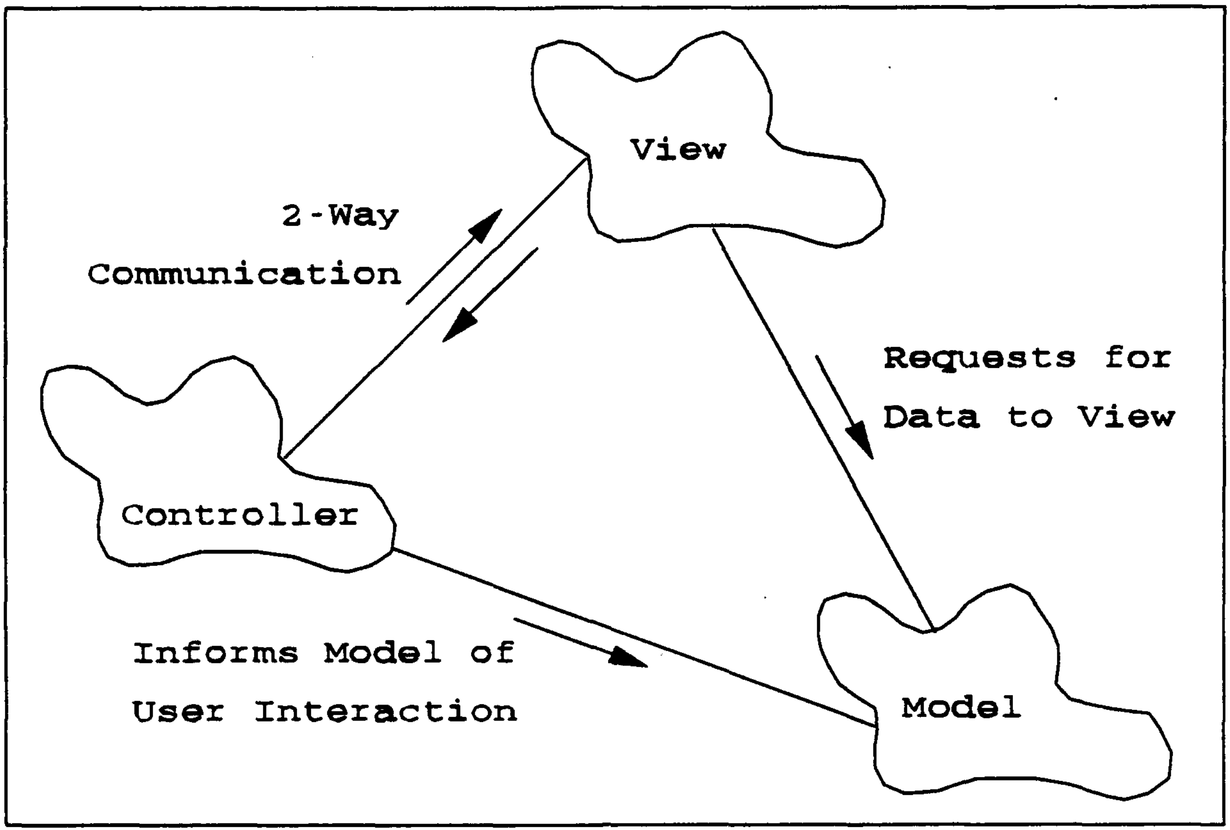


Figure 3 Client / Server Relationships in the MVC Paradigm

Each of the view/controller pairs may be seen as useful abstractions for commonly used interactive displays.

2 LRDC's Generic Graphics Tools

It has been shown that Smalltalk's window classes provide useful abstractions for interactive displays. However, the types of displays, and the means by which a user may interact with them, are rather limited. The LRDCGP extends these capabilities enormously. This is accomplished in two ways: new objects that represent complex graphic entities, and new MVC classes that know how to draw these entities.

2.1 GraphicObject and GraphicElement Classes

At the root of the expanded graphics capability are the classes GraphicObject and GraphicElement. Instances of GraphicObject may represent very complex graphic entities composed of multiple subentities, which are instances of GraphicElement. Each GraphicObject may have an optional identifier. Its composite GraphicElements know about their optional identifier, geometric shape, level, and color. The level of a GraphicElement indicates how opaque graphic elements look when they are displayed one on top of the other. If the GraphicElement is a TextElement, it may also know about its font.

The most powerful aspect of these classes is that they know how to draw themselves when given a pen and character scanner to do it with. Just how powerful this can be will be shown in the next section.

2.2 LRDCGP's MVC Classes

As was previously explained, when an instance of GraphPane is asked to update itself, it requests the information to be displayed by sending its contents message to its model. The model will either send nil, indicating that the pane should copy the area under it, or will return an instance of one Form, or one of its subclasses.

If we were to want to display a geometric shape in this GraphPane, we would have to use a pen to draw it on a Form, and return that form to the GraphPane. This might not be so complicated, but what if we were to draw a complex object, such as a pie chart. Our application would be required to know how to draw pie charts. This might not be so bad, except that all applications that were to display pie charts would be required to have this same knowledge.

Pie charts are used as an example here because they are such a common abstraction. Their behavior is well defined, and known to many people. It would make for much more reusable code if there were a PieChart class. Then, instead of having applications know how to draw pie charts, we could have the instance of PieChart draw itself. However, this would require specializations to the already existing MVC classes.

This is precisely the tact taken by LRDCGP. This section will describe LRDCGP's MVC classes, most particularly its specialized view and controller. With these classes, any object that can return a GraphicObject representation of itself may be drawn. In this way, the application programmer is abstracted away from the mechanics of drawing, and may focus instead on the characteristics of the objects to be drawn.

2.2.1 Model: GraphicEditor

The model using the LRDCGP's MVC is much the same as the one using Smalltalk's standard MVC. In both cases the application programmer is abstracted away from the mechanics of capturing mouse events and interpreting them as a selection at a particular point on the pane. The programmer is also abstracted away from how to display the object in a pane. But in the case of the standard Smalltalk graphic application, the application must know how to manipulate pens and character scanners in order to draw on the instance of Form to be displayed. In the case of the LRDCGP's graphic application, the program need only know how to obtain an abstracted GraphicObject representation of the objects to be displayed. The LRDCGP is a natural exention to the MVC paradigm. It allows an application programmer who is already adept at building windowed applications in the MVC paradigm to easily add graphics in a fashion that fits seamlessly with the underlying windowing mechanism.

```
graphicObject
graphicObjects
graphicObjectForFrame:
graphicObjectsForFrame:
```

1 Order of Precedence for Graphics Generating Messages

Notice how the low the level of abstraction would be if the application needed to create GraphicObject instances for the objects to be drawn. However, there is a much more attractive solution if these objects are useful abstractions that may be useful in other applications. Object oriented design would dictate that it should be these objects themselves that create their GraphicObject representation. In this way the application programmer need only be concerned with the state of those objects, and allow those objects to determine their own graphic form accordingly.

Thus, the model for an instance of GraphicsPane should return either an instance of GraphicObject, or a collection of instances of GraphicObject, or an object or collection of objects that respond to one of the messages in text box 1. Notice that these messages are in order of precedence. The message that will be sent will be the first one in this list that the object responds to. It should be obvious that the corresponding methods should return either a single instance of GraphicObject or a collection of instances of GraphicObject. If the method takes an argument, it is the rectangular region, or "frame", in which it is to be displayed. This may be useful for having the application scale the objects in a fashion appropriate to its needs.

GraphicEditor is an example of a software model that uses the LRDCGP's MVC extensions. It may be used as the model to a single paned application. It can be used as the display and database part of a drawing program when used in conjunction with the graphics editing tools. It may also be used as the model for one GraphicsPane in a multi-paned application. There are many reasons in which this design would be desirable. Most often, the GraphicEditor would be specialized to make particular types of displays. For instance, the LRDC has had to build a CircuitEditor specialization which knows about instances of the class *Circuit*, and how to interrogate Circuits for their states. This added information allows color coding which is used for educational purposes. This CircuitEditor is a useful abstraction that could be reused in several different applications.

This next section describes the use of the CircuitEditor as a model of just one pane in a multiple paned application. This requires the use of the dependency mechanism. In our present implementation, the dependency mechanism was not used. However, this will soon be changed. The change should not be too difficult, and would allow our graphics tools (the hypergraphics in particular) to be reused much more easily.

In this situation, there would be an application class that would be the model for the TopPane. This class would store the state of the application, and orchestrate the interdependencies between panes. Let the CircuitEditor's GraphicsPane have the name *circuit*. The CircuitEditor is informed that it is a view for the application. Just as when a pane is told that it is a view for a model, the CircuitEditor makes itself a dependent of the application. When the application wants to update its graphics pane, it sends the message *'self changed: #circuitEditor'*. The dependency would find that the application has the CircuitEditor as a dependent with name *circuitEditor*. The CircuitEditor's circuitEditor method would get its state from its owner / model, i.e. the application class that is using it. The application would inform the CircuitEditor of which Circuit to display. The circuitEditor method would then ask to update its circuit GraphicsPane by sending the message *'self changed: #circuit'*. The dependency mechanism would now request that the GraphicEditor's GraphicsPane update its contents with what is returned by the method circuit. The circuit contents method could either return the instance of Circuit to be displayed, or it could ask the Circuit for its graphic representation. It could then ask its owner to color this graphic representation, and return the modified data.

This example shows one way in which the LRDCGP both supports and encourages a very high level of abstraction. This makes for both simplified, readable classes, as well as components cf applications that may be reused in other applications.

2.2.2 View: GraphicsPane

GraphicsPane is a specialization of GraphPane. With respect to its display capabilities, its implementation is very simple, since its only task is to as its pen to draw the information returned from its model onto an instance of Form. Once this has been done, the GraphicsPane can show its form in much the same fashion that its super class would have.

However, the user may also choose to interact with the objects obtained from the GraphicsPane's model via their graphic representations. As will be discussed in the next section, this is done via active areas, which are instances of Region. Therefore, when the GraphicsPane obtains the object or objects to be drawn from its model, it asks each object for its active region by sending the message *activeRegions*. The result of which is given to the receiver's dispatcher.

2.2.3 Controller: GraphDispatcher and RegionDispatcher

If the objects to be drawn are simple graphics, then Smalltalk's standard GraphDispatcher adequately meets the purpose. However, if the user is expected to interact with the objects of which the GraphicObject instances are a graphic representation, the GraphicsPane's dispatcher will need some added capabilities. As was mentioned in the last section, the RegionDispatcher is given the active areas for the objects being displayed by the GraphicsPane. This is isomorphic to the client / server relationship between the standard Smalltalk MVC components, as shown in Figure 3.

When the user clicks over an spot not covered by an active Region, the change method associated with that pane is sent to the GraphicsPane's model. This is exactly what would have happened with the standard Smalltalk MVC classes. However, if the user clicks at a point that is covered by an active Region, that region informs its model of this interaction.

dispatcher
model
rectangle
hiLit
up
down
highlight
lowlight
userData

2 Region's Instance Variables

The Region class is extremely flexible. Text box 2 shows the variables for instances of Region.

In this context, the Region's *dispatcher* will be set by the GraphicsPane's RegionDispatcher, when it is asked to add the Region.

The Region's *model* is the object that is to respond to reported changes in the Region's state. When an object is requested for its active Region by a GraphicsPane, that object usually becomes the model for the Region. Thus, when the RegionDispatcher processes user input, the region sends a message to its model, appropriate to the type of input.

The user may interact with a Region by clicking in an area specified in the Region's *rectangle* IV. Though the IV is called "rectangle" it may be any geometric shape that respond to the predicate message *containsPoint:*.

The IV *hilit* is private, and is set to true if and only if the cursor is inside the Region's rectangle.

If the Region is informed of an up-button event, the region will send its up function message to its model. This message is stored in the *up* IV.

If the Region is informed of a down-button event, the region will send its down function message to its model. This message is stored in the *down* IV.

By default, when the cursor enters an active Region, this is indicated by performing the default highlight function, which is to reverse video the Region's rectangle. When the cursor exits the Region's rectangle, the Region performs its default lowlight function, which is to return the rectangle to its original form. These actions may be overridden by setting the Region's *highlight* and *lowlight* Ivs to messages that are to be sent to the Region's model in these situations.

The Region's *userData* IV is used to store important information about that Region that the programmer has thought of, but which is not already captured by the Region and its model. However, in the present context, these Regions only come into being by a GraphicsPane requesting the Region of an object to be displayed. Since the purpose of the Region is for a user to interact with an object via its graphic representation, that object is the model for the Region. If that object is implemented as a truly encapsulated class, then it should know everything about its internal state. Therefore, there is rarely any useful information that an object may want to put on its Region's userData property list when it is asked for its Region.

2.3 Specialized GraphicsPen

As has already been described, instances of GraphicObject already know how to draw themselves when given an instance of Pen and an instance of CharacterScanner to do it with. The only reason to have the GraphicsPen specialization of Pen is so that is knows how to send the messages listed in text box to the objects it is to draw. Since these messages return instances of GraphicObject, the GraphicsPen can then ask each of the GraphicsObject to draw itself with the receiving GraphicsPen. Since some of these GraphicObjects may have TextGraphicElements, the GraphicsPen must also be part CharacterScanner. Since Smalltalk does not have multiple inheritance, GraphicsPen is implemented as a Pen with a CharacterScanner, initialized onto the same form. (Actually, it is presently implemented as a pair of such Pen/Scanner objects. It was a quick hack that allowed easy highlighting of active Regions. It is soon to be fixed).

2.4 Editing Tools

The parts of the LRDCGP already described provide incredible flexibility. However, a common need is for drawing programs. Whereas a paint program "paints" with a brush, changing pixels beneath it, a drawing program creates and displays perhaps complex graphic entities, that may be composed of any number of elements. These elements may be text or may be geometric shapes. From this description, it is obvious that the objects can be represented by instances of GraphicObject and its composite GraphicElements. However, an interface is required to use these in a drawing program.

Dan should probably polish this portion, if for no other reason than to have it reflect reality.

The LRDC has developed a library of classes that have been used by a permanent menu class. This menu class was built with extensibility in mind, and has been used to implement the interface to a graphics editing package. The graphics editing classes may be specialized to edit not only graphic objects, but also other objects with a graphic representation. For instance, these editing capabilities could be specialized to allow a person to graphically build an instance of Circuit.

These classes can obviously be used to build very sophisticated applications for the generation, inspection, and manipulation of an endless variety of objects. The document, Graphics Based Construction Tools by Dan Peters should be read for a discussion on how to use these tools in applications.

It is important to note that the LRDCGP facilitates the development of complex graphics tasks by providing tools for every stage of the application's decomposition. The application programmer need only focus on the behavior of the application as a whole, and the objects to be displayed. The LRDCGP provides the tools for displaying, creating, and editing these objects. Each of these capabilities may be incorporated by the seamless connection of one of the tools to the application.

2.5 Printing Graphics

Many platforms provide utilities for the creation of graphics files that may be sent to a printer. Smalltalk has utilities for the creation of PostScript files from the contents of forms. Remember that instances of GraphicObject may be asked to draw itself by giving it an instance of Pen and an instance of CharacterScanner. Also remember that these drawing devices may be initialized to a Form. Since the utilities take their input from the same display devices to which the drawing devices give their output, it is easy to use these utilities print graphics generated by the LRDCGP. Although the LRDC cannot take credit for the utilities, it is a very useful feature that the product of the LRDCGP fits so seamlessly with the existing classes that it can exploit existing utilities.

3 Specializing LRDC Graphics Tools

As has been previously discussed, LRDCGP is most powerful when it is used to display objects that know how to create GraphicObject representations of themselves. The LRDCGP includes many examples of such objects. These objects are felt to be of potential use in any number of applications.

This section gives an examples how a programmer can build a library of useful abstractions that know how to create graphic representations of themselves, and how to use and reuse them easily in multiple applications. The example will show how experienced MVC application programmers can easily incorporate graphics with a methodology which is consistent with their non-graphics windowed application development methodology.

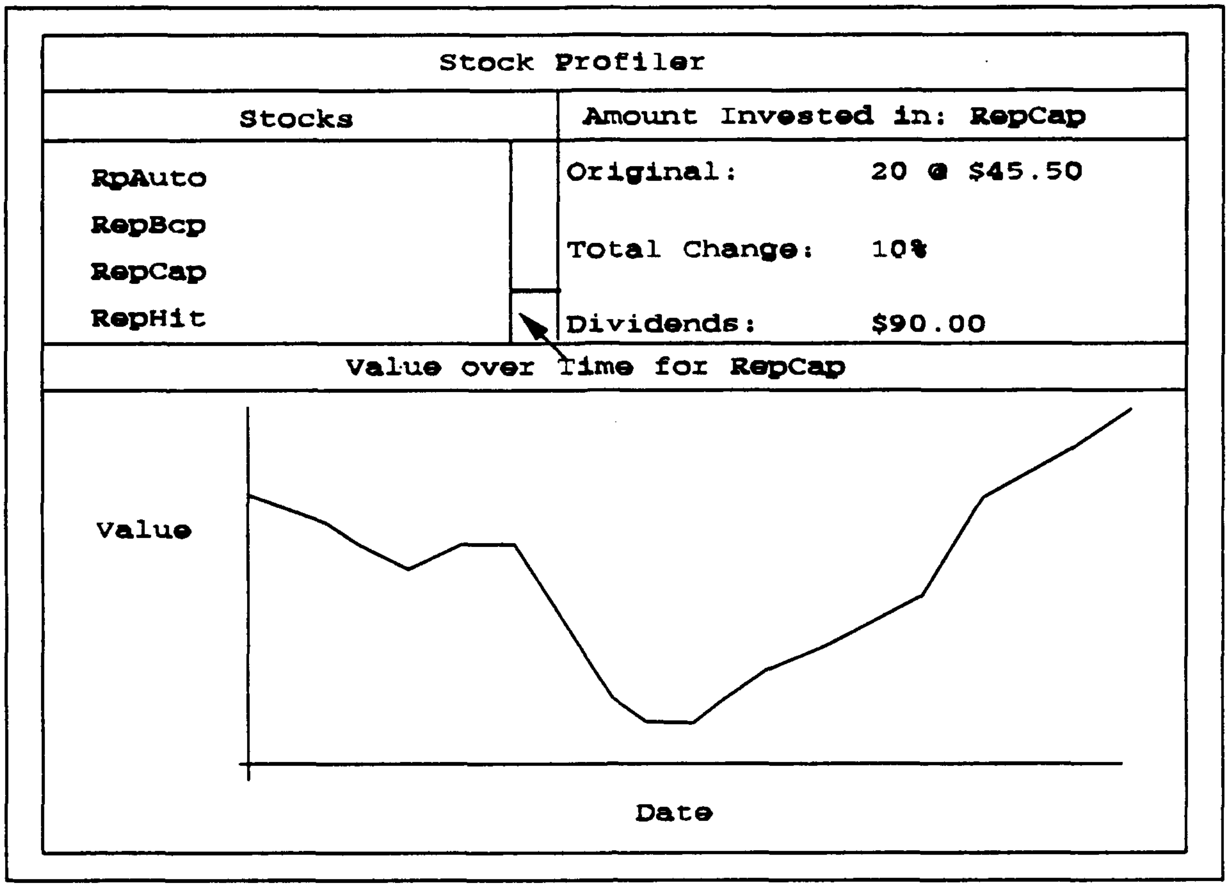
3.1 Stocks Portfolio Profiler Example

Let us suppose that we have an application that displays the performance of stocks over time. Our application might look something like the window in Figure 4.

The implementation would require very little code from the programmer. Essentially, we would need an application that would supply the contents and change methods for the subpanes shown in Figure 4, and classes for the graphs to be displayed.

Since the purpose of this document is to describe the LRDCGP, we will only describe how the graphics in the bottom pane are displayed. In this example, the bottom pane is used to graphically depict different aspects of a person's stock portfolio. Since this GraphicsPane will be used to display charts, let it be named *chart*. Thus, when this GraphicsPane wants to update itself, it will get its contents by sending the message *chart* to its model, an instance of *StockProfiler*.

We could have the instance of StockProfiler laboriously figure out how to create an instance of GraphicObject. This would not be so bad, since some object must ultimately be responsible for this task. But, since Charts are useful abstractions, let us instead encapsulate the behavior of this abstraction as a new class instead. That way, other applications could reuse these Chart classes.



Stock Profiler

Stocks

RpAuto
RepBcp
RepCap
RepHit

Amount Invested in: RepCap

Original: 20 @ $45.50

Total Change: 10%

Dividends: $90.00

Value over Time for RepCap

Figure 4 Simple Stock Portfolio Profiler

3.1.1 Chart Classes

The Chart classes have only been partially implemented. They would require only one uniterupted day to finish.

LRDC's Graphics Package includes several commonly used types of charts used for many applications. Examples of these chart subclasses are drawn in Figure 5. The class hierarchy is shown in text box 3. These include smooth and discrete continuous graphs on two axes for showing the correlation of one variable as a function of another, bar charts most commonly for comparing of two variables, and pie charts for showing relative proportions.

3.1.2 Updating the Profiler's Graphics Pane

In the first example, the GraphicsPane has been used to display a graph of a selected stock's performance over time. Figure 4 shows how an instance of Graph was used to represent the value of the stock at discrete intervals.

Text box 4 shows a sample trace of what happens when a user selects a new stock to display.

Notice how Figure 4 shows an instance of ListPane in the top left corner. This ListPane would list all the stocks that a user could choose from. Since clicking over one of the stocks itemized would constitute selecting a stock to display,

```
art
    BarChart
        HorizontalBarChart
        VerticalBarChart
    Graph
        SmoothGraph
    PieChart
```

3 Class Hierarchy for LRDC's Chart Library

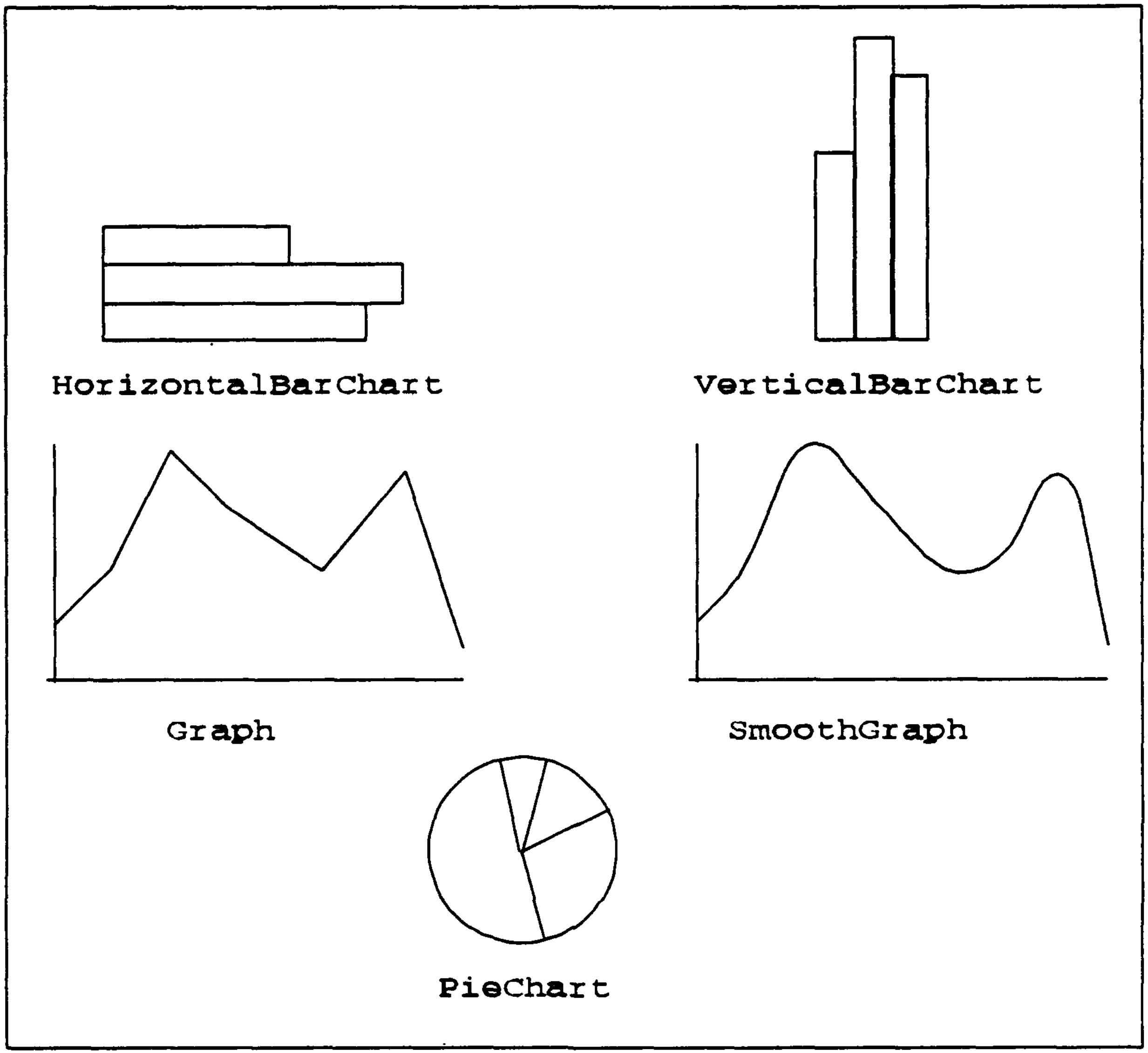


Figure 5 Example Instances of Subclasses of Chart

its change method will be called *stock:*. The method stock: would set the application's state to know which stock the user selected, and then would inform the application that its chart pane has been changed by sending the message, *self changed: #chart.*

The Dependency mechanism would now inform the GraphicsPane named chart that it should update its contents. It would do this by setting its displayInformation IV to the result of sending the message *chart* to its model.

The application would know its state now was set to view the RapCap stock. The application could query its database for RapCap's performance over time and create a new instance of Graph, giving it, say, a dozen (X, Y) pairs. This instance of Graph would be returned to the GraphicsPane.

The GraphicsPane would now ask its GraphicsPen to draw the instance of Graph on its backup form.

The GraphicsPen would ask for the Graph for its graphic representation.

The Graph would return a GraphicObject which would be composed of a collection of GraphicElements, each of which would have a line as its geometric shape. The "sum" of these lines would describe the axes and the function of value against time.

The GraphicsPen would ask the GraphicObject to draw itself using the receiving GraphicPen.

```
1  StockProfiler>>stock: 'RepCap'
   stock := 'RepCap'.
   self update: #chart

2  GraphicsPane>>showWindow

3  StockProfiler>>chart
   Returns a Chart.

2  (GraphicsPane>>showWindow)
   displayInformation := resulting chart.

3  GraphicsPen>>
    graphicObjectsForInfo:
      displayInformation
    frame: anAreaOfScreen

3  GraphicsPen>>
    graphicObjectFor: anObject
    frame: anareaofscreen
   Returns a collection of GraphicObjects representing
anObject by sending messages in text box 1.

2  (GraphicPane>>showWindow)
   graphicObjects :=
    resulting collection.

3  GraphicsPen>>
    drawGraphicObjects:
      graphicObjects

4  GraphicObject>>
    drawWith: aGraphicsPen
    writeWith: aScanner

5  GraphicElement>>
    drawWith: aGraphicsPen
    writeWith: aScanner
   Colors aGraphicsPen and aScanner.
   Asks geometricShape IV to draw itself with
aGraphicsPen.

3  GraphicsPane(GraphPane)>>
    showWindow
```

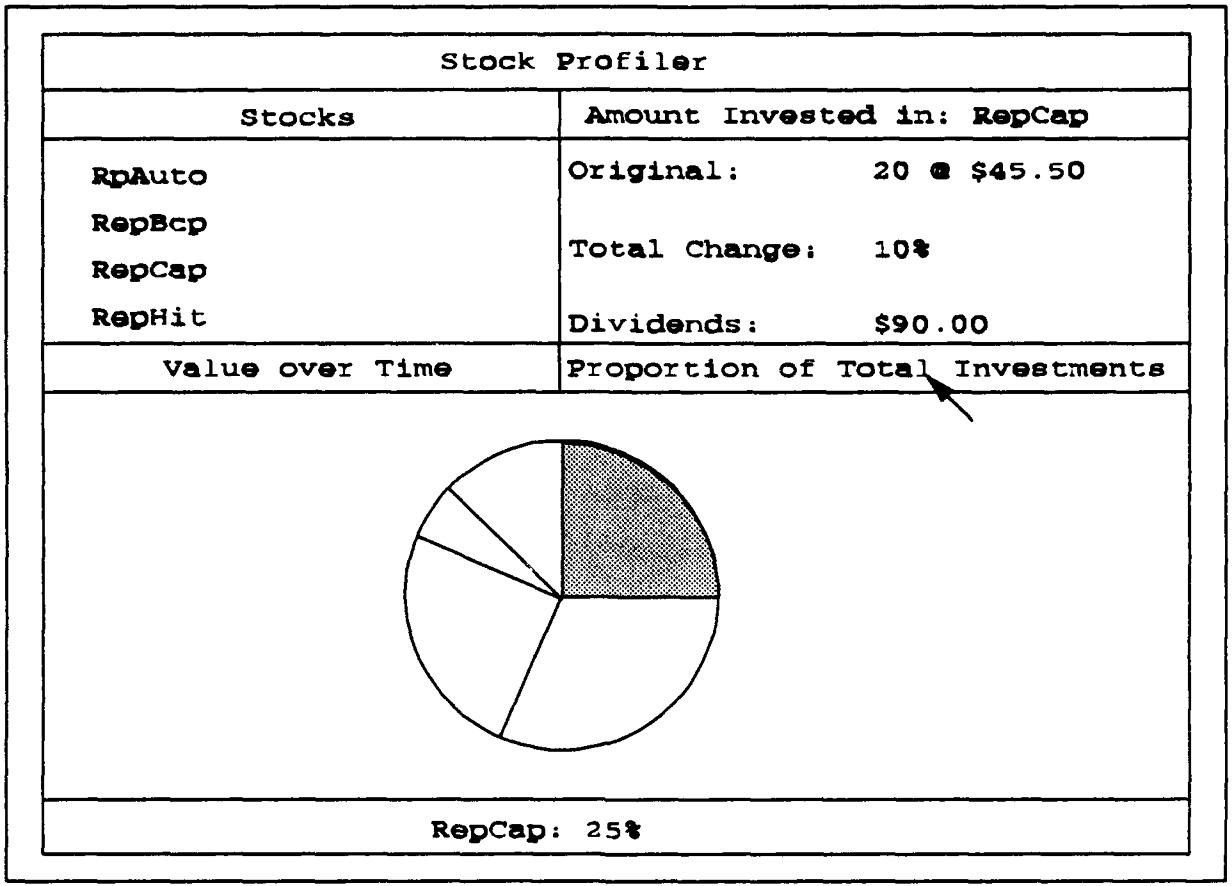
4 Selecting a New Stock to Display

The GraphicObject would ask each of its GraphicElements to draw itself with the GraphicsPen.

Each GraphicElement would set the GraphicPen's color and thickness, and ask its geometric shape to draw itself with the GraphicPen.

3.2 Adding new Graphics to the Profiler

The purpose of this section is to show just how easy it is to modify an existing application or to build a new application by reusing classes that implement useful abstractions, and which know how to draw themselves.



Stock Profiler

Stocks

Amount Invested in: RepCap

RpAuto

RepBcp

RepCap

RepHit

Original: 20 @ $45.50

Total Change: 10%

Dividends: $90.00

Value over Time

Proportion of Total Investments

RepCap: 25%

Figure 6 Stock Portfolio Profiler with Investment Percentages

In this example we will extend the capabilities of the Stock Profiler by using one of the existing Chart subclasses. The new Stock Profiler will allow the user to select if he or she is to be shown the performance of the selected stock over time, or the value of that stock as a proportion of the total portfolio's value. The StockProfiler class must now be able to record this selected state. Let us give StockProfiler the IV *chartMode*, which may have one of two values: *value* or *proportion*. If chartMode is set to value, then an instance of Graph is to be displayed in a fashion similar to that shown in Figure 4. If chartMode is set to proportion, then an instance of PieChart is to be displayed, as shown in Figure 6. This PieChart would show the proportion of all the stocks in the portfolio, and color the selected stock.

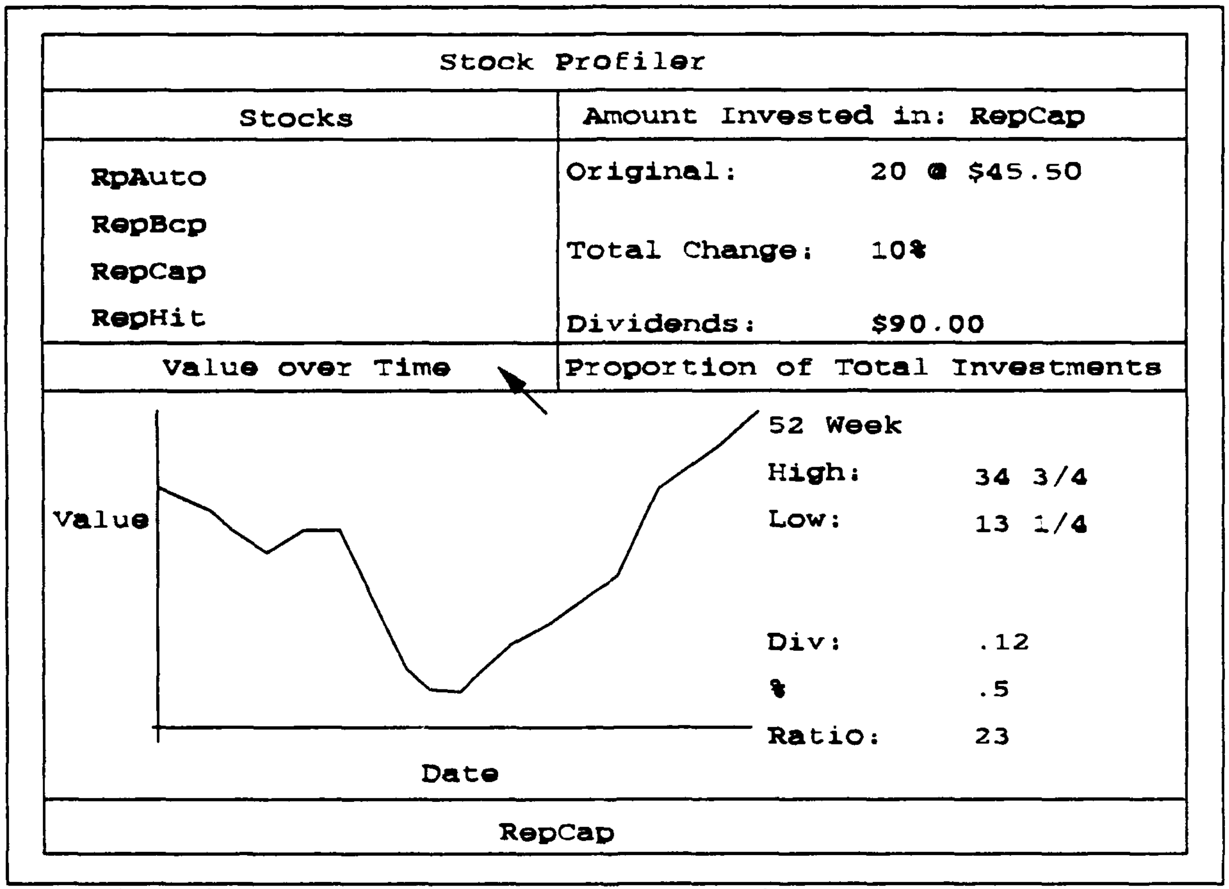
Above the chart GraphicsPane in Figure 6 are two buttons used to set the chartMode state of the application. These buttons are easily implemented as instances of ListPane with only one selection. The button on the left has the change selector *value:*, and name *value*. The method *value* corresponds to the value pane's contents, and returns the string *'Value over Time'*. The button on the right has the change selector *proportion:*, and name *proportion*. The method *proportion* corresponds to the proportion pane's contents, and returns the string *'Proportion of Total Investments'*.

The two change methods simply set the StockProfiler's state to either value or proportion and then send the message *changed: chart* to itself. The dependency mechanism will now inform the chart GraphicsPane to update its contents in a fashion similar to that shown in text box 4. The method chart which returns the for the chart GraphicsPane method must now initialize a new Graph or a new PieChart depending on its chartMode. Since points and proportions are easily described, the programmer is abstracted away from how to actually plot and display the points or pie sections.

3.3 Objects with Complex Graphic Representations

The Report / ReportField and TextBox classes have not yet been fully implemented.

So far, the chart GraphicsPane of Figure 4 and Figure 6 has been used to display instances of Chart. The characteristics of the chart abstraction are well known and well defined. However, sometimes other information might be helpful. Information that might not be easily encapsulated into a reusable Class.



Stock Profiler

Stocks	Amount Invested in: RepCap
RpAuto RepBcp RepCap RepHit	Original: 20 @ $45.50 Total Change: 10% Dividends: $90.00
Value over Time	Proportion of Total Investments

52 Week

High:	34 3/4
Low:	13 1/4
Div:	.12
%	.5
Ratio:	23

RepCap

Figure 7 Stock Portfolio Profiler utilizing instances of Report

Let us consider another enhancement to our Stock Profiler. In Figure 6, the chart GraphicsPane gives an impression of RepCap's performance, but the specifics are not immediately obvious. Perhaps we would like to also display to display added information from today's NYSE. An example of this is shown in Figure 7.

The text on the left will be unchanging. The values that correspond to the text description must be updated and filled into the fields on the right. Such template filling is common in database programming, where the template is often called either a 'Form' or a 'Report'. Since

Smalltalk already has a Form class, we will create a Report class. Reports will be used to display reusable classes that know how to create graphic representations of themselves, such as Charts, as well as text and fields, which will obtain their value from the Report's owner each time the Report is updated.

3.3.1 TextBox Class

We have already seen how to implement and use classes that represent concepts that have a meaningful graphic representation. Sometimes we just need to present text. Let us create a class, called TextBox, that would know all the necessary information for how and where to display a piece of text. This information would include the text, its font, its location, whether or not it should have a visible border, its color, the border's color, the border's fill color. In fact, this is analogous to the information that is known by the TextElement subclass of GraphicElement, except that TextBoxes are used to store information, whereas TextElements are also used to draw and write themselves. Since all the graphics information is inherited from GraphicElement, TextElements have been implemented to use a TextBox for the added information. Thus, a TextElement for a GraphicObject may easily created from a TextBox by simply including the TextBox in a new instance of TextElement. (The common features between the TextElement and the fictitious TextBox have not, as yet, been separated).

3.3.2 Report Classes

The entity that we want to encapsulate may be separated into two parts: a containing object with which applications interface, and the contained objects, known only to the container.

3.3.2.1 Report

Reports are the implementation of the previously mentioned containing object. When an application wants a graphic representation of the objects known by the Report, a request is sent to the Report, not directly to the objects it contains.

The Report must store a position for each object to be shown. These are stored as an OrderedCollection of pairs in the IV *displayObjects*. One element of the pair is the object's position. The other element is the object to be asked for its graphic representation. When the Report is asked for its graphic representation, it collects the graphic representations of its stored objects, and positions them in the locations stored in the displayObjects IV.

3.3.2.2 ReportField

Consider the type of objects that may be stored in a Report. The simple case is when the object to be drawn is static, i.e. its graphic representation never changes, so the object need not be updated. Examples of this would include '52 Week', 'High:', and 'Low:' in Figure 7.

A more interesting situation occurs when the object's state must be updated before it is to be displayed. We need a new type of object in this situation. One that knows how to update its object's value. Examples of this would include the values that have been filled in after the 'High:' and the 'Low:' TextBoxes, i.e. '34 3/4' and '13 1/4'.

ReportFields have the following IV's:

- o owner
- o name
- o updateSelector
- o value

Just as SubPanes get their contents from their models, ReportField's get their contents from their owners. When a ReportField is informed that it should update itself it sends the message stored in *updateSelector* to its owner. The updateSelector takes the ReportField's name as its sole argument. The result is an object that knows how to create a graphic representation of itself.

When a ReportField is asked for its graphic representation, it first updates itself, and then it requests the result, stored in the IV value, for its graphic representation.

For the sake of consistency, all objects know how to create a graphic representation of themselves. The method graphicObject is inherited from the class Object. The default action is to create a TextBox whose textual contents are the result of sending the message *printString* to the object. This default may be meaningless, but will prevent applications from crashing. However, this simple solution does mean that instances of String and Number may be returned as the value from the ReportField's owner in response to the updateSelector.

In the example of Figure 7, consider the '34 3/4' after the 'High:'. As has been explained, the text 'High:' is constant, and is simply stored as either a TextBox or a String, if the default font, etc., are considered to be sufficient. The field that responded to the user request for the 52 week high, however, is dynamic. Its value had to be updated from some database. To do this, we would create a ReportField with the name fiftyTwoWeekHigh. Let the owner be a database which responds to the updateSelector *updateVariable:*. Then, when the ReportField is asked to update itself it performs the following: *'value := owner perform: updateSelector with: name'*. This would be equivalent to '*value := owner updateVariable: #fiftyTwoWeekHigh*'. The database would look up the value for the variable fiftyTwoWeekHigh, and return the number 33 3/4.

What about default display characteristics, like font, flush left, etc.?

Best not to answer this until after Report classes have been implemented.

4 Hypergraphics

We have just looked at an example of how an application may be quickly built to graphically display stock performance. This example showed how an application can easily incorporate complex graphics while being abstracted away from the mechanics of graphics displays.

Another popular use of graphic displays is for interactive graphics. This document refers to these capabilities as hypergraphics, when in fact they are more accurately described as hyper media. This is because the LRDCGP's hyper graphics allow the system to use video if available. The LRDC has created classes that allow the developer to control a video disk player. These tools include a laser disk play and graphic overlay board server, as well as a graphic remote control. The actual implementation will not be discussed here. It is only necessary to know that these facilities are provided by a server class called LDP (for LaserDiskPlayer). It responds to a small set of requests for simple operations such as advancing forward and in reverse, pausing, and searching to a specific frame, etc..

One example of interactive graphics is the popular Hypercard product. This product is commonly used on PCs for quickly implementing user interfaces and instructional software. Such products are commonly known as "authoring languages" since they abstract the developer away from the details of implementation, and allow the developer to concentrate on the intent of the application. In fact, the developer is so abstracted away from the implementation, that many Hypercard developers are self-taught, with no other programming experience.

As with all high-level languages, the weaknesses of authoring languages are the direct result of their strengths. High-level languages are often designed to directly support a particular paradigm. For instance, there are many 4th-Generation languages on the marked that are explicitly designed to support the relational model for database programming. The main advantage is that the programmer does not need to implement the data structures to represent these relations at the low level that, say, a C programmer would. When these languages are used for appropriate applications, the database program can be written in a very easily read fashion in very few lines of code. However, these languages are designed very specifically to their target applications. Such languages, for instance, would not be very suitable for rule-based programming.

Similarly, authoring languages allow a developer to author interactive environments very quickly. This makes them very attractive to both experienced and inexperienced programmers alike. However, these languages are rather limited in their expressiveness. An experienced programmer may find that the useful abstractions supported by the authoring language thwarts his or her creativity, since the programmer has no control over the underlying data structures. The programmer is limited to the abstract data-types directly supported.

LRDC's hypergraphics classes are intended for the use of proficient programmers. They are intended to be both reusable and extensible. Just as the Model-View-Controller paradigm abstracts the developer from the mundane tasks frequently used in interface design in a flexible fashion, the LRDC's hypergraphics classes facilitates interactive graphics programming, while leaving the developer the full expressiveness of the Smalltalk language and object hierarchy.

Although the hypergraphics classes were designed to be extremely reusable and extensible in many domains, they were originally designed so as to facilitate building an interface to a large and complex piece of electronic diagnostic equipment as part of the Sherlock project. A small proportion of this diagnostic equipment (known as a "test station) is shown in Figure 8. We will use this as one example of how to use the LRDC hypergraphics classes.

The diagnostic equipment can be separated into different areas, each with its own purpose. Examples would include a digital multimeter and an oscilloscope, which are detailed in Figure 8. Since each area can be physically removed from the entire test station, they are called "drawers".

If you have ever looked at an oscilloscope, you will realize that each drawer may be quite complex, and that at any one time, you would probably only want to see a specific area, such as the "Time per Division" and "Volts per Division" controls. You could perhaps navigate your way around by use of nested pop-up menus, the first offering the full views of each of the test station's drawers. After selecting such a drawer, a submenu would allow you to "zoom" into a functional area of that drawer, etc.. At any point, your options would depend on your previous decisions. Figure 9 Shows a graphic representation of the resulting decision tree.

4.1 Hypergraph as a Database

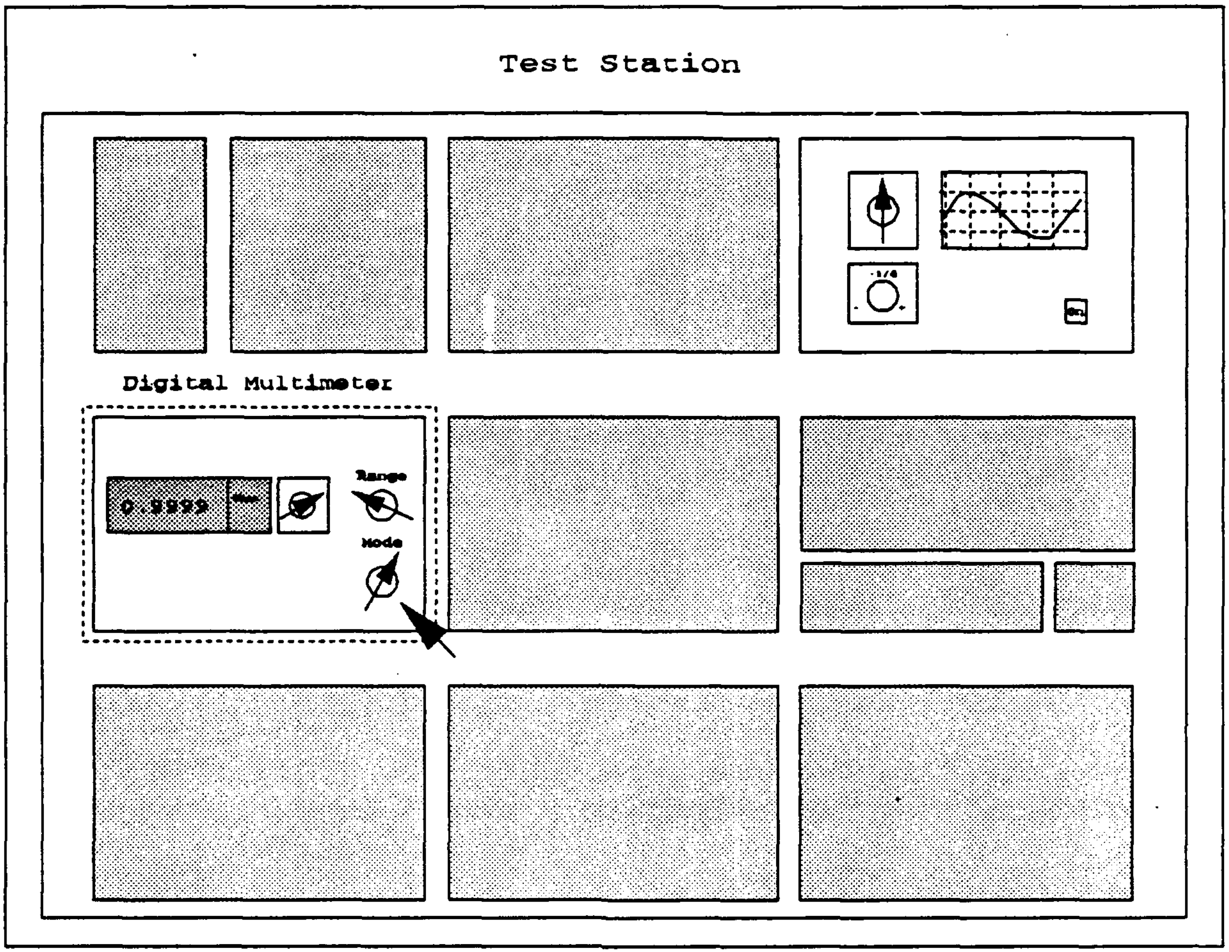


Figure 8 Simplified Version of a Test Station

The hypergraph is the data structure that contains all of the data required to describe any hypergraphic world. is a graphic representation of the world we might need to describe for our example.

The graph is a database that stores the following:

- the different views
- what views may be visited from any view now being displayed
- what devices the user may interact with when displaying any particular view.

4.1.1 HypergraphFrames

For historical reasons, each graphic view of a hypergraphic world is called a "frame". In our example, we will want a frame for the entire test station, a frame for each drawer, and as many frames as are needed to supply close-up views that the user may need to "zoom" into.

4.1.2 Hypergraph

The arcs in the hypergraph represent the relation "can-zoom-to" between frames. For example, when viewing the entire test station, the user might want to zoom into a close-up view

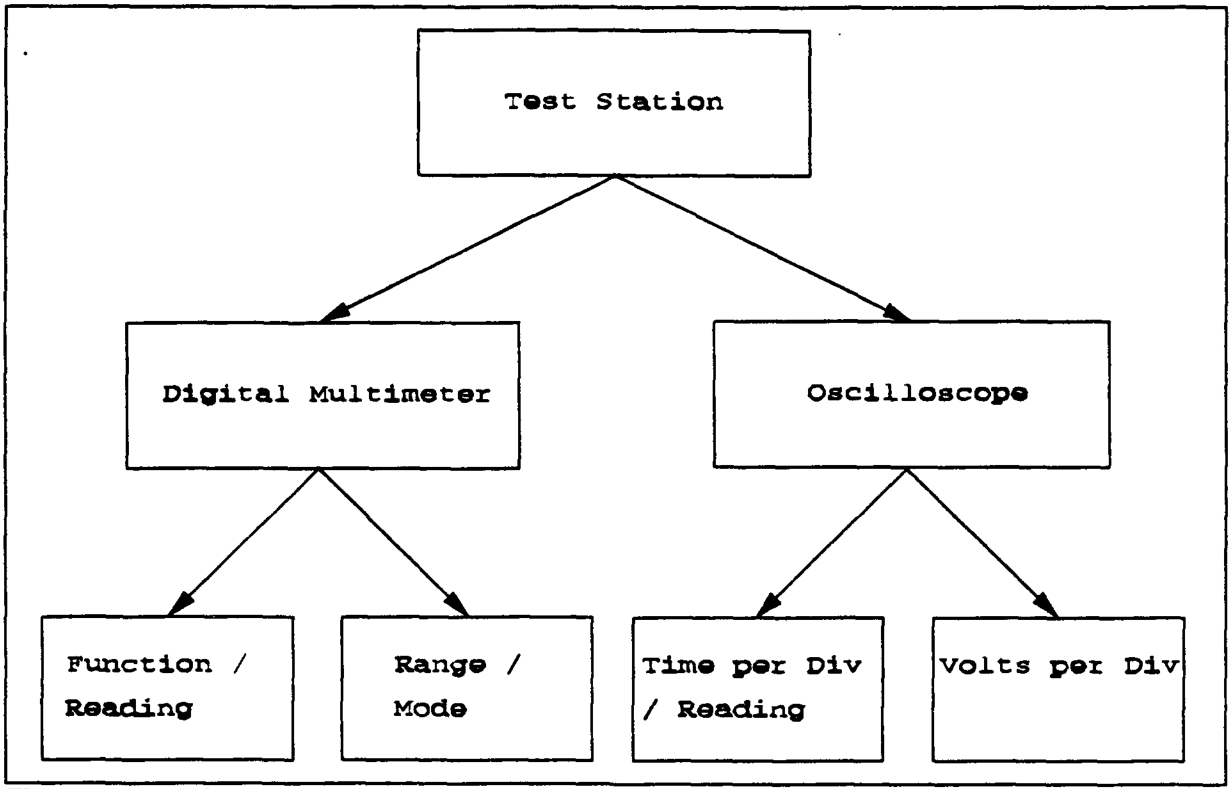


Figure 9 Simplified Map for Navigating the Test Station

of the oscilloscope. This is exactly the same as the pop-up menu interaction described previously, and drawn in Figure 9.

4.1.3 Devices

Device and its subclasses are very powerful encapsulations of objects that know how to create state dependent graphic representations of themselves, and how to change their state based on user interactions.

Text box 5 shows the class hierarchy structure for the LRDC's library of Devices. The classes written in italics are abstract classes. No instances of these classes exist, only instances of their subclasses.

Some of these subclasses of Device have names that would indicate that they have very specific uses. This is a historic accident, which occurred before the exact behaviors of desired devices were fully understood. The names were a useful, though now outdated, mnemonic. Indeed, the behavior of several of these subclasses have become indistinguishable from each other. These problems will be remedied in future releases.

- *Device*
    - ChangeDataBaseFrameDevice
    - *DisplayDevice*
        - *Indicator*
            - DigitalIndicator
            - *MultiIndicator*
                - HorizontalMultiIndicator
                - VerticalMultiIndicator
            - SplineIndicator
        - PushButton
        - *RangeDevice*
            - RotatingKnob
            - WrapDevice
    - MultiVideoFrameDevice
    - Part

5 Class Hierarchy of LRDC's Device Library

As has been stated, a Device's graphic representation is based on its state. Its state includes not only its 'value' but also other display information, such as 'lit' or 'unlit', 'on' or 'off', 'open' or 'closed', or whatever other states are useful which may modify how the Device's state is displayed, if it is displayed at all. Exactly how the Device is to be displayed is also determined by whether or not the system is using video. If there is video, the Device often returns no graphic object, allowing the video to show its state. However, if there is no vide, the hypergraphic application may be blank, with no clues as to where the user will find the Devices, should the user want to interact with them.

Though many of Device's subclasses share the same display methods, they differ in how the user may interact with them. When a Device is activated, the device will respond to the message 'activeRegions' by returning whatever Regions are required to allow a user to interact with the Device. As was explained in the section entitled, 'Controller: GraphDispatcher and RegionDispatcher', GraphicsPanes ask objects not only for their graphic representation, but for the object's active regions. This is most effectively used when the object creates regions for which it is the mode. Then, when the region is informed of user interaction, it may send appropriate messages to the object. In this situation, user interactions are interpreted as requests to change the Device's state in some meaningful, predictable fashion.

4.1.3.1 ChangeDataBaseFrameDevice

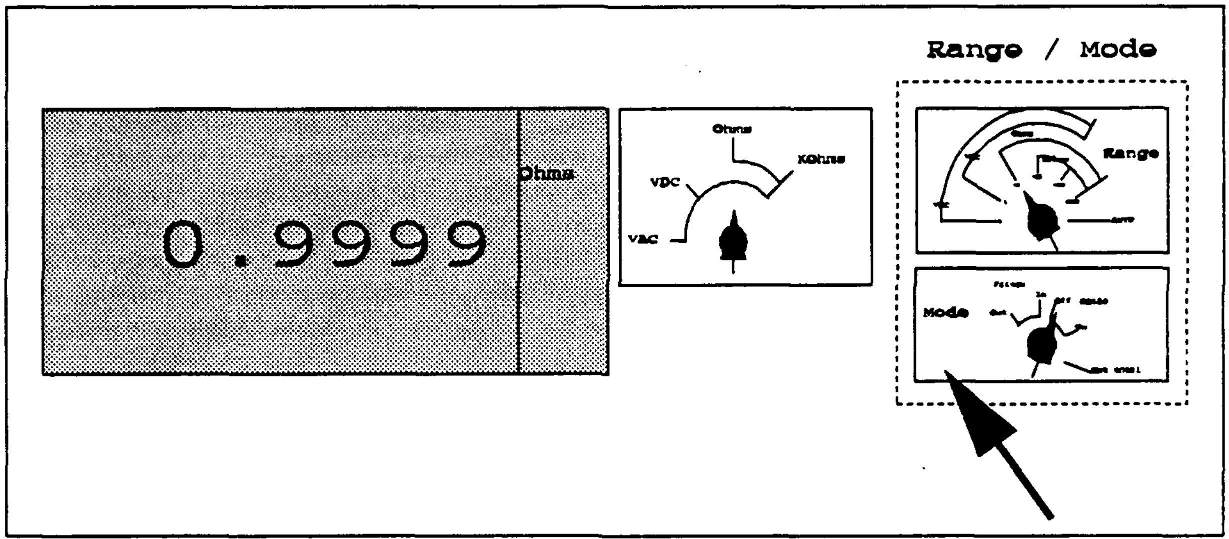


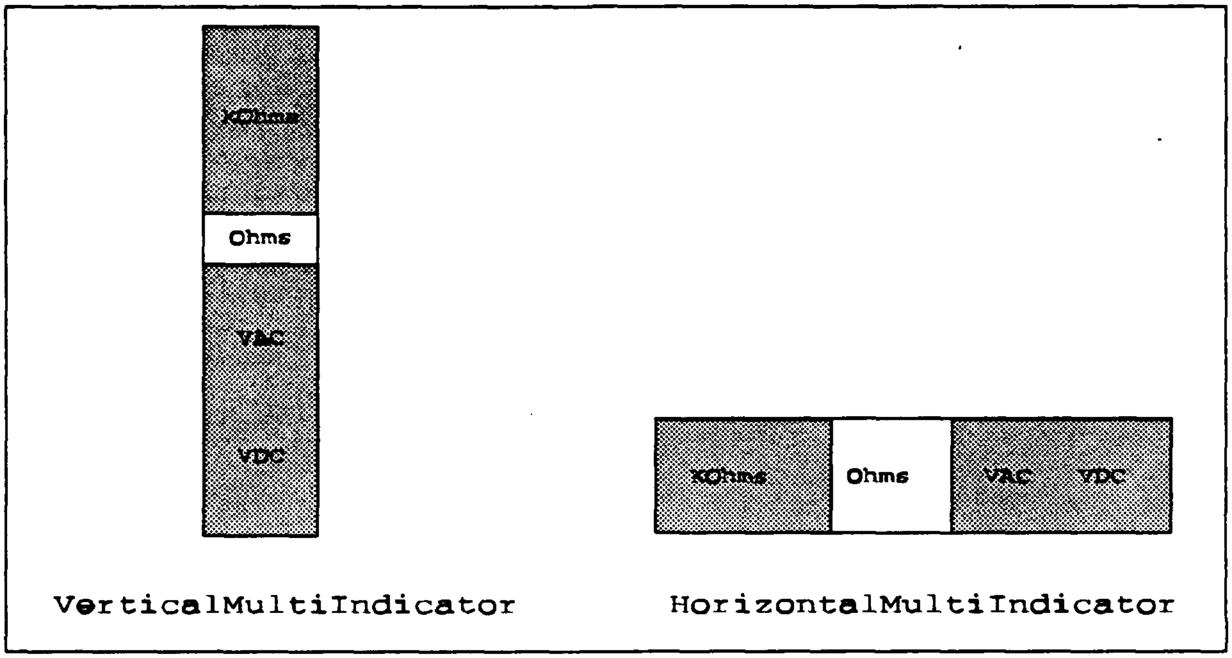
Figure 10 Close-up View of the Digital Multimeter

This is the device that provides the user with the ability to view different HypergraphFrames. If you look closely at Figure 8 you will see that the cursor has entered an area around the "Digital Multimeter" portion of the test station. This area is covered by an instance of ChangeDataBaseFrameDevice. The device creates and active Region over this area, and by entering its region the border is illuminated in blue and displaying the region's name above it, as shown. This indicates to the user that depressing the button over this area will have the effect of "zooming" into a close-up view of this region, as shown in Figure 10. Similarly, the cursor in Figure 10 is shown entering the "Range / Mode" area of the digital multimeter. A mouse click over this area would "zoom" the user into the close-up view of the "Range / Mode" area of the digital multimeter, as shown in Figure 12.

4.1.3.2 DigitalIndicator

Figure 10 shows a close up view of the Digital Multimeter. One of the most important parts of modeling the multimeter's interface was to provide a digital read-out area to display the current reading on the multimeter. This is implemented by use of an instance of DigitalIndicator. This name is somewhat misleading, since these indicators can probably display any string, not just floating point numbers. When made active, instances of DigitalIndicator can be either 'lit' or 'unlit'. When their state is 'unlit' they do not display their value.

4.1.3.3 MultiIndicator



There are two types of MultiIndicator, those which display vertically, and those which display horizontally. Examples of these are shown in Figure 11. When made active, MultiIndicators act as buttons which move through the allowable states in a ring fashion. Thus, in the example of Figure 11, if the user were to click on either example, the indicator would change its state from 'Ohms' to 'VAC', thus lowlighting the 'Ohms' area, and highlighting the 'VAC' area.

4.1.3.4 SplineIndicator

SplineIndicators are used to display curves. In fact, they could display any object that knows how to generate a graphic representation of themselves. An example of this is shown in the top right hand corner of the full view of the test station, in Figure 8. Here, an instance of SplineIndicator is used to model the display area of an oscilloscope.

4.1.3.4 PushButton

Instances of PushButton have only two display states, 'on' and 'off'. They can take on only one value, which is their name. When a PushButton is 'on' it displays its name in the center of its display area. When it is 'off' it indicates this by showing its display area in black, if there is no video, or by not displaying itself at all, if there is video.

4.1.3.5 RangeDevice (actually RotatingKnob)

The name RotatingKnob is a historic accident, which will change in future releases.

It is often useful to consider objects that have a range of values. These devices have a maximum and a minimum value. RangeDevices give two active regions when asked. One allows the user to decrement the Device's value, and the other allows the user to increment the Device's value. When a user tries to increment the Device's reading beyond the Device's maximum, or decrement its value below its minimum, the Device's value remains 'pegged' at its current value.

4.1.3.6 WrapDevice

Instances of WrapDevice are identical to instances of RangeDevice. However, when a user tries to advance the receiver's state beyond its end values, the value 'wraps' around using modulus arithmetic. This is implemented using the abstract datatype Ring. Actually, it may still use the method *rem:*, though the class Ring has been implemented.

4.1.3.7 MultiVideoFrameDevice

Figure 12 shows a close-up view of the "Range / Mode" area of the digital multimeter. This view has two instances of MultiVideoFrameDevice.

Let us first consider the situation where only the mode knob is to be modeled. It has five possible states, corresponding to the five positions in which its knob may be turned. Figure 12 shows the cursor over the location of the 'Ratio On' selection. If the user were to now depress the button, we would want the video player to move to the video frame which displays the knob rotated so that it is pointing to 'Ratio On'. The knob would now be in the 'Ratio On' state. Thus, MultiVideoFramDevices must have a mapping between each of its states, and the video frame that displays the device in that state. It must also have one active region for each of its possible states so that the user may set the device's state. These regions should be positioned in a meaningful fashion as shown in Figure 12.

When the HyperGraphPen is asked to draw the HyperGraphFrame, it must advance the LDP to the appropriate video frame. To do this, the HyperGraphPen asks the HyperGraphFrame for the video frame. This would be relatively easy in our simple example. Since there is only one Device that knows about what video frame to display, the HyperGraph would ask the Mode for its appropriate video frame. The Device would see that it is has the state 'Ratio On', and would look up the appropriate video frame from the prestored mapping.

Now let us consider the situation where there are two instances of MultiVideoFrameDevice being displayed in one HypergraphFrame. Let us look at the example in Figure 12. The Range Device's state has been set to '10'. The Mode Device has been set to state 'Ratio Off'. If the user now wanted to set the Mode Device to state 'Ratio On', he or she could click of the corresponding Region, which has been positioned over the 'On' part of the video display. The Mode Device would now be informed by the Region that the user clicked over an area that correspond to 'Ratio On'. In this case, the Region's property list has to be informed of this region's purpose. This contradicts an earlier statement! Modify the earlier remark to be less strong. The Mode Device will now set its state to 'Ratio On'.

Now, when the HyperGraphPen asks the HyperGraphFrame for the video frame to display, the HyperGraphFrame finds two instances of Device that know about what video frame to display, both the Range Device and the Mode Device. When the HyperGraph would ask these Devices for the appropriate video frame for their state, they must be the same frame, since we want to

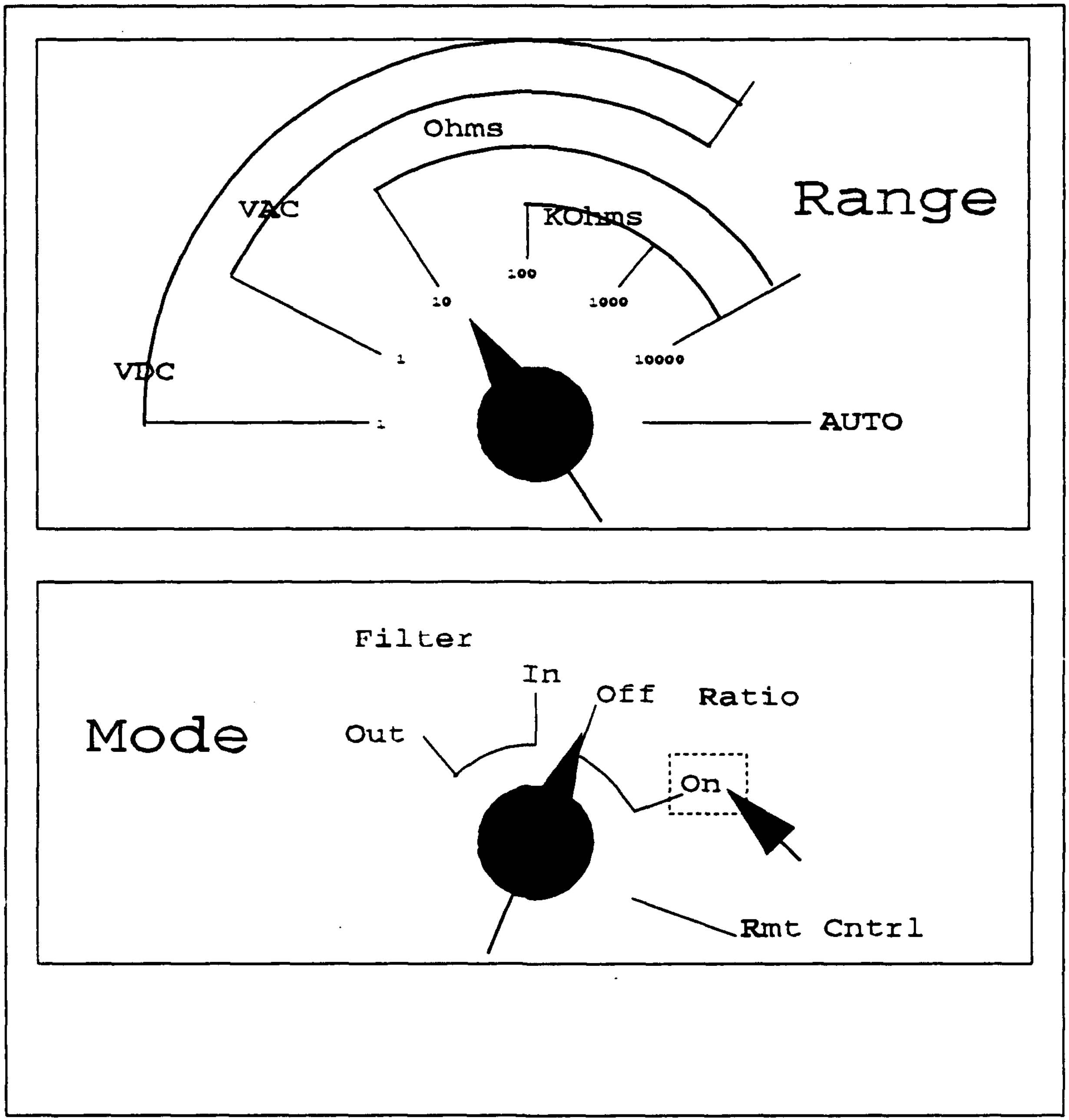


Figure 12 "Range / Mode" area of Digital Multimeter

show one and only one video frame. There are, in fact, 7 video frames which would be legal for the Mode Device in this state. They would be when the Mode Device had state 'Ratio On' and the Range Device simultaneously had one of the states '1', '2', '10', '100', '1000', or 'AUTO'. The Mode Device would see that it is has the state 'Ratio On', and would look up the set of appropriate video frames from the prestored mapping. When the Range Device is asked for its legal video frames, it would see that it is has the state '10', and would look up the set of appropriate video frames from its prestored mapping. The only video frame that would actually display the appropriate state of both Devices would be found by the intersection of the two sets.

4.1.4 Extending the Device Classes

The Device's that LRDCGP's library provides entirely encapsulate their behavior in terms of their graphic representation, their states, how to create Regions to capture user interactions, and how their states change due to such user interaction. All the developer has to do is to select the most appropriate Device class, and to use it. With this approach, the developer does not need to know how to program at all!

Other authoring systems, such as HyperCard, allow the developer to encode the Device's behavior in response to user interaction. This has the slight disadvantage that a developer has to be well versed in HyperCard's authoring language, HyperTalk. However, HyperTalk is a very specialized language which is very easy to learn. This scheme gives a lot more control to the developer. Since each Device could have its own code, the only inherited characteristics would be the enumeration of states, and the display characteristics given the Device's state. Though there are other disadvantages to this approach which will be discussed later, the HyperCard approach certainly does allow greater versatility. This is a topic that will require greater research.

4.2 LRDC Hypergraphics Display Tools

The LRDC's Hypergraphics tools are simple extensions to the LRDCGP already discussed. Just as the LRDCGP classes were the natural extension of the existing windowing platform, the LRDCGP's hypergraphics classes are a natural extension of the previously documented LRDCGP classes. An application programmer who is adept at using the Smalltalk windowing classes will find adding hypergraphics to their application to be a simple task.

4.2.1 Model: HypergraphicDisplay

The GraphicEditor example of an application using the LRDCGP was introduced in the section entitled 'Model: GraphicEditor'. The class HypergraphicDisplay is a specialization of the GraphicEditor class. It too may be used as a stand-alone application. Remember that GraphicEditor is most useful as part of a drawing program when used in conjunction with editing tools. Similarly, HyperGraphicDisplay is of most interest when used as the display mechanism for hypergraphic editing tools to be described later.

Of course, HyperGraphicDisplay may be specialized to model some useful stand-alone application. However, as with GraphicEditor, HyperGraphicDisplay is most powerful when it is incorporated into an application as the model of one of that application's GraphicsPane. The HyperGraphicDisplay would now be a reusable, extensible software module that could be incorporated into future applications. In this situation, the HyperGraphicDisplay is a complete reusable view mechanism of instances of HyperGraph. The owning application is thus entirely abstracted away from the requirements of a hypergraphic application.

4.2.2 View: GraphicsPane

Since HyperGraphics classes respond to the messages in text box 1, the GraphicsPane previously described may still be used.

4.2.3 Controller: RegionDispatcher

Although applications using simple graphics may use either the GraphDispatcher or the RegionDispatcher, all hypergraphics applications must have RegionDispatchers for their GraphicsPanes. It is the RegionDispatcher that interprets user interactions with Regions and sends appropriate messages to the Regions. These Regions are used to allow the user to manipulate Devices.

4.3 Specialized HyperGraphicsPen

HyperGraphicsPen is a specialization of GraphicsPen. The only new capability that they have is that they know how to ask objects for the video frame that should now be displayed. They then ask the LDP utility to display this video frame.

4.4 Editing Tools

I'm not sure if Dan finished this implementation.

The section entitled 'Graphics Based Construction Tools' introduced the LRDCGP's tools used for editing objects graphically. The graphic editor class has been specialized so as to allow a non-programmer to create new Hypergraphs, with the ability to create, delete, and edit HypergraphFrames and the Devices on such frames.

4.5 Using LRDC Hypergraphics

The hypergraphics classes of the LRDCGP may be used wholly or partly in any application. However, the LRDC enthusiastically endorses one particular approach that encourages the development of maintainable applications. The essence of this approach is the identification and isolation of the simulation and the interface components of an application.

4.5.1 Server: Simulation Interface Application

As was discussed in the section entitled, ", LRDCGP's Devices encapsulate not only their state and their appearance, but also how they respond to user interaction. This has the advantage that the developer need not program, but merely select the most appropriate existing Device class. The problem occurs when there is no such match. This might lead one to the HyperCard approach, where the developer has full reign over the Device's response to user interaction. If the LRDC were to give its developers the same freedom, one might just leave the entire power of Smalltalk to the developer's disposal. The developer would now be able to encode the behavior of each Device individually. The developer could even choose to encode interdependencies between objects. However, to do this would blur the distinction between the interface and the underlying simulation.

The unification of interface and simulation has been used in many systems, including some popular ones built on Smalltalk. These will often use the Dependency mechanism to broadcast effects to other objects which depend on a Device whose state has just changed. Though this has lead to powerful simulations, there is now way to easily use the same interface with a different simulation engine. Similarly, it is very difficult to use a different interface for the same simulation engine. In the LRDC's context, we chose to change our simulation from a black-box model of circuit components to a deeper model, utilizing the mathematical abstraction of a directed graph. This was done effortlessly, since the simulation was entirely separated from the interface. If the LRDC were to choose to add the ability to customize the behavior of individual instances, a specialized language would have to be implemented to prevent the blurring of interface and simulation. This scripting language would enforce the rule that objects could only change their own state.

Hypergraphs are therefore only databases, since the do not embody any aspect of the simulation. Let us first consider the situation where there is no simulation. When a Device is informed of a user interaction, it may change its state appropriately. When the HypergraphicDisplay updates itself, the only visible change will be in that Device, since the state of no other Device has been changed.

4.5.2 Client: HypergraphicsDisplay

The simulation application may be thought of as a software model of some activity. Like a finite state machine, it knows how to respond to changes in the state of the model. The hypergraphics is just an interface to that simulation, interpreting user interactions, and informing the simulation of changes to the states of devices. An application may be used to find a mapping between Device names and the corresponding simulation objects.

In the previous section, the HypergraphicDisplay was shown to be able to create a simple interactive environment. This environment, however, does not provide global effects between Devices. Normally, however, hypergraphics are used as a view onto a simulation. In this situation, the hypergraph must update its data for the HyperGraphFrame being displayed. For this reason, when a HyperGraphFrame is asked for its graphic representation, it asks its HyperGraph to update the values of the Devices to be displayed from the model to which it is an interface.

Text box 6 shows what happens when a user clicks on a power off switch. After the Device updates its value to 'CLOSED', the simulation is informed of the change. Since this switch provides power to all other devices in the simulation, they are now all given the state 'OFF'. When the HypergraphicsDisplay is asked to redisplay the HyperGraphFrame, each Device is asked to be updated from the HyperGraph's database, which in this case is the simulation. Each Device in turn is set to 'OFF', the graphic representation of which is usually a blank.

In this way, the effect of a Device's state change is kept local, with respect to the interface, but global effects may be displayed by informing and interrogating the simulation of state changes.

1. Device is informed of user interaction, and updates its state.

2. Device informs the HyperGraph of which it is a data element of its new state.

3. HyperGraph informs its owner of the Device's new state.

4. HypergraphicDisplay finds simulation object that corresponds to the Device name and updates its value.

5. Owning application updates its GraphicsPane.

6. HyperGraphFrame is asked for its graphic representation.

7. HyperGraphFrame asks each Device for its graphic representation.

8. Device asks its HyperGraph to update its state.

9. HyperGraph updates the Device by asking its owning application for the new values.

10. The owning application asks the simulation to update the Device.

11. Each Device returns its state dependent graphic representation.

12. HyperGraphPen asks the HyperGraphFrame for the current video frame to display.

13. HyperGraphPen tells LDP to go to correct video frame.

14. GraphicsPane displays new contents

6 Updating a HyperGraphFrame

Index

Sherlock Technical Document

KeyPad:
A Versatile Interaction Device

DRAFT

Dan Peters
University Of Pittsburgh
LRDC
July 29, 1991

I. Introduction.

This document serves to describe the structure and use of a KeyPad mechanism written in Smalltalk v/286. This mechanism is useful in a variety of contexts in which users must choose some item from several. The classes which are a part of the KeyPad package provide many functions useful for constructing custom selection objects. Objects which fall into this category include items such as permanent menus, pull-down menus, button boxes, and tool palettes. The classes provided with the KeyPad package make constructing and using any of these items easy. In the text which follows, we first take a look at exactly what is available in the package. In this section we will examine the various classes which are a part of the KeyPad application and how they are used. We will then proceed to more technical matters as we discuss the various interaction protocols which are required in order to use these facilities. Next, we will explore the possibilities for extension to these classes via inheritance.

II. Available Facilities.

In this section, we briefly describe the facilities currently available for constructing several kinds of interaction objects. There are basically two type of objects useful in these endeavors. The type of object is embodied by the class KeyManager and its subclasses KeyGroup and KeyPad. KeyManager is an abstract class providing behavior and semantics common to other types of Key management classes. It is here that the interaction protocol shared by KeyPad and KeyGroup is defined and implemented. The subclasses KeyPad and KeyGroup inherit this protocol from KeyManager and add behavior more specific to their function. A KeyPad is useful in situations where a stand-alone interaction facility is called for. For example, if some application required a menu which could be moved about on the screen independently of the application's other windows, then a KeyPad would prove useful. Another place where a KeyPad object is useful is in situations where the interaction mechanism must have persistence independent of a specific window. KeyGroups, on the other hand, are useful whenever a selection mechanism is to be included as part of some other window. KeyGroups will define and manage a subset of the window's panes, reporting to its owner any interaction taking place inside this area. These types of objects are very useful in implementing objects such as menu bars with pull down menus, or tool palettes like those that are usually found along the periphery of graphics programs' windows.

The second class of objects that we use to build interaction devices is the Key class. This class is an abstract class with many subclasses which implement several different kinds of input mechanisms. In this hierarchy are included items like Buttons, PictoKeys, MenuButtons, and colored varieties of each. An instance of class Button provides very simple behavior. When it is "pressed" via a mouse click, it simply causes (via its KeyManager) a message to be sent to the application for which it is acting. This message is defined by the application and will include as a parameter an id for the Button which is also assigned by the application. This mechanism provides for a very rich set of possible behaviors, as we will shortly see.

PictoKeys are objects like those usually found in tool palettes. Unlike Buttons, which display text inside their view, PictoKeys can display bitmapped graphics inside their view. These object are quite useful in a variety of contexts. For example, if we had a graphics program and we wanted to provide a Line-Draw tool, w could put up a KeyPad with a PictoKey on it, and this PictoKey could display a picture of a line.

Another type of Key which is available is the MenuButton. These items behave in a fashion similar to that of pull-down menu items. They behave the same as a Button (indeed they inherit from Button) except that when they are "pressed" they pop up an application defined menu, allowing the user to choose one of the items in the menu. When a choice is made, the application will be informed of this event via an application defined message. As a parameter to this message, the selector corresponding to the menu choice made will be provided.

Together, objects inheriting from classes KeyManager and Key provide a very diverse and useful range of interaction possibilities. Because of the way these items were designed, Key managers can have any kind of keys on them in any combination. This implies that a keyManager can manage several \different types of keys at once, and this provides even more flexibility.

III. Interaction Protocols.

Keys and KeyManagers were designed to provide a maximum amount of flexibility for both use and extension. In this section, we explore the flexibility of use by describing the procedure by which these objects are created and the protocol by which they are used.

The basic requirements for creating keys are defined by the superclass Key. All keys require two pieces of information. The first is the key's id. An id is any object which the application finds useful in identifying the key. For example, if pressing a certain key is supposed to cause the application to invoke one of its methods, a good id for the key could be the name of that method. Then when the application is informed of keypad activity, it can check to see of the item received as a parameter is the name of one of its messages. If so then it can signal itself to perform the method. In this way the id can be used not only to identify the key, but also to help define the semantics of a given keypress. The second piece of information that should be provided for all keys is the key's contents. This item represents what will be shown inside the key's view. Buttons accept a string as their contents and will display the string's text inside the view. PictoKeys similarly accept a form containing the picture that they are to show.

Subclasses of Key may require additional information beyond an id. Specifically, MenuButtons need to be given a menu to display when they are pressed. The MenuButton subclass OwnedMenuButton must be made aware of two additional items: an object which can provide the button with a menu to display and the name of a message to send to that object in order to obtain the menu. Additionally, each of the key subclasses has a colored counterpart. These keys provide the same semantics but display themselves in user defined colors instead of the default. These colors are provided by the application via the messages **foreColor:** and **backColor:**. A summary of the messages requires to create keys appears in figure 1 below.

KeyManagers are a little more complex to create in that they require several pieces of information. KeyManagers are created and initialized via the message **for:**. The parameter to this method is the object that the keypad is to act for or report to. Once an instance is obtained in answer to this message, several other messages must be sent to the instance. First, the instance needs to be given the name of a single parameter message to send to the application whenever activity occurs inside the view. The parameter provided will be the id of the key that was activated. Next, the instance need to know about the number and arrangement of the keys it controls. These pieces of information are provided by sending the message **addKeysIn:inRectangle:**. The first parameter should be an Array. The elements of this Array should themselves be instances of class Array containing all of the keys in a given row. The second parameter should be a rectangle indicating the size (in number of keys) of the keyed area.

Four example, if we wanted the KeyManager instance to operate with respect to a 4x3 array of keys, we would send it the message named above with the required array of keys as the first parameter and a Rectangle with origin 0@0 and corner 3@3. (Note that the keymatrix is zero offset). We can also use this message to specify subsets of keys to appear in the overall matrix. The overall size of the keymatrix is specified by the message **matrixSize:**. This is also a requirement. Finally, the message **keySize:** must be sent with a Point parameter indicating the size (in pixels) of each of the key's views. No checking for overlap is done. Once all of this information has been provided, we can open the keypad with the message **open** or **openIn: aRectangle**.

IV. Structure.

Having explored the means by which KeyManagers and Keys are created and used, we now turn to a discussion on the actual structure of these items. In doing this, we can see how these facilities provide for a maximum of design flexibility and ease of extension. We will also illustrate these ideas by reviewing the design of the MenuButton Subclass.

Keys are viewed as objects that can react to interactions with the mouse. They provide a view onto some information via a SubPane. Any type of SubPane may be used to view a Key. For example, ListPanes are used to view Buttons whereas GraphPanes view PictoKeys. The type of Pane viewing the Key is an intrinsic part of the Key and an initialized Pane of the required type is provided by sending the message **keyPane** to any Key.

In addition to providing a view onto some information, Keys are also responsible for communicating mouse events to the KeyManager that they are a part of. Specifically, each Key has implemented a method called (usually) **pressKey:**. This method will then take any Key specific actions such as popping up a menu or highlighting its view. Following this, a message is sent to the KeyManager with two parameters: the id of the pressed Key and the name of a message to send to all of the other Keys controlled by the KeyManager. It is not necessary for any of the other keys to implement the method named by this second parameter. This second parameter is useful in establishing state coupling between keys without making the Keys themselves aware of the existence of other Keys. An example of this could be a "SHIFT" key which causes some or all of the other keys controlled by the manager to change their view to some shifted state. The default message sent is **refresh** which causes Keys to update their views. This basically defines the dynamic behavior of Keys, and when coupled with the information in the preceding section, specifies the overall behavior of Keys.

KeyManagers are used to coordinate the activities of keys and to communicate Key activity to some application object. Again, the protocol is extremely simple. As mentioned above, when a Key is pressed, it sends a two parameter message to its Manager. When the Manager receives this message, it first dispatches a message to its application with the id of the activated Key as a parameter. The name of this message is defined during KeyManager creation as described above. When the application answers, the KeyManager will then dispatch the message whose name was received from the key to all of the other keys being managed. That's basically all there is to it, since the interface was designed to purposefully be quite simple. All of the other activities supported by these objects are concerned with view management or maintenance of internal state. The simplicity of this interface frees the application from most of the drudgery associated with pane management and event handling. Instead, it only needs to take action based on the reception of a Key id, and can remain completely ignorant of any of the details concerning the events leading up to this reception. As a trade, the application designer must create and initialize the required objects, but this is very simple and well worth the trade off.

We now illustrate the ideas explored above by reviewing the design of both a custom menuing device and a custom Key type. Suppose it was required to construct a menu-driven graphics program which provided all of the usual graphics facilities like shape drawing, file I/O, cut/copy/paste, etc. We now concern ourselves with providing the menu from which all of the choices will be made. First, we recognize that many of the operations involved are similar in some way. For example, drawing circles and squares are very similar activities in that they cause something to be drawn. Also, adding, deleting, moving, and coloring shapes are similar activities in that they all operate with respect to some piece of graphics information. This observation suggests that in designing the interface, it would probably be useful if we could group similar items together in some recognizable fashion. This is generally done via pull-down menus on a menu bar in most applications. In our case, however, suppose also that it is desirable in the interests of generality, to make the selection mechanism independent of the application. We might finally decide on creating a "free-floating" menu object containing the items we wish to display. As far as grouping similar items together, two choices immediately suggest themselves. First, we could color similar items the same color. This could work, however, if we have many choices, the menu could become rather large and unwieldy. A better approach would be to have a few general choice items available and when the user selects one of these items, pop up a menu of items which correspond to the selection. For example, we could have an item on the selection object called "Edit..." When the user selects this item, a pop-up menu could appear listing things like cut, paste, color, etc. This is the choice that we will make. Now suppose that we are examining the facilities provided by Key and KeyManager and are shocked to discover that no key providing a pop-up menu upon selection is available. (MenuButtons provide this facility, however let's pretend for a moment we don't have them.) What is to be done? Well, let's make our own.

First, we decide to make the new type a subclass of Button because the view will display a string like "Edit..." and most of the behavior we seek is already provided by Button. In inspecting Button, we arrive at the conclusion that it will be necessary to make primarily two changes to achieve the behavior we seek. First, we will need to be able to pop-up a menu. We would like for this menu to be provided by the application when the MenuButton is initialized. Therefore we will need a method, say, **menu:** along with an IV to hold the menu. All that remains is to change the selection semantics so that the menu is popped up when the selection is made. As stated above, the **pressKey:** method is responsible for handling selection activities for a Key. Therefore this method will need to be redefined in our subclass. In writing the code for these two methods, we would probably arrive at something like the code found in figure 2.

<<<<<<<   FIGURE 2

```
menu: aMenu

    "{Dan Peters: Jul 18, 1991 11:01:16}"
    "
        PUBLIC
        ======

        Assign aMenu to the receiver.
    "

    menu := aMenu.!

pressKey: aString


    "{Dan Peters: Jun  4, 1991 12:10:54}"

    "
        LIMITED PUBLIC
        ==============

        This method should be called only
        by the receiver's view.  It informs the
        keypad of activity in the key's view.
        aString is ignored.
        This type of button pops up its menu and
        waits for input.  When this input arrives,
        its associated selector is returned to
        the keypad.  If the selection is nil, then
        the key just updates itself without notifying
        the keypad that an event has occurred.
    "
    | response |

    response := menu popUpAt:
        (Cursor offset:
            (keyPane frame corner)).
    response isNil
    ifTrue:  [ ^self refresh ]
    ifFalse: [
        keyPad perform: padUpdateMessage
                with:  response
                with:  #refresh
    ].                    >>>>>>>>>>>>  <<MAKE INTO COLUMNS>
```

That's all there is to it; we now have "pull-down" menu keys. When one of these items is selected, the associated menu will be place overtop the selected key and the selection from the pop-up menu will be communicated to the application via the KeyManager. If no choice is made, the Key merely refreshes itself and does nothing.

Now that we have the required keytype, we need to decide on some issues concerning the interaction between the KeyManager and the application. First, we should choose how many keys we will need and how they should be arranged. Looking at the requirements of the problem, (as it was proposed for actual implementation) we arrived at basically four types of possible selections: **Create Objects..., Edit Objects..., File..., and View....** We will therefore need our KeyManager to manage four MenuButtons displaying the above items. We chose to arrange these MenuButtons in a 2x2 Array. Each MenuButton also requires a menu to display. Since the application will receive the selector associated with a menu selection from the KeyManager, it is worthwhile to consider what we should use as selectors. Since the items in **Create Objects...** all cause some Smalltalk object to be created, a good choice for the selectors for this menu could be a list of Class names corresponding to the items in the list. When the application receives notice that a selection has been made, it can use the parameter to find out if something should be created by examining its type, and it can find out what type of object is to be created by examining the parameter's value. The other MenuButtons all seem to be responsible for the invocation of some function provided by the Application. Therefore, we would probably want to use method names as selectors, and have the application dispatch the named message when a message name is received.

Having made these decisions, we can now proceed to implement the interaction object. To do this, we will need basically three things: a method to create and initialize the KeyManager, a method to create and initialize the Keys, and a method to respond to event detection. An example implementation for these methods appears in figure 4.

```
<<<<<<<<<<FIGURE 4

createMenu

   "{Dan Peters: Jul 26, 1991 11:18:00}"
   "
        PRIVATE
        =======

        Create a keypad which the receiver can use to communicate with
        the user.  Also, create the keys to appear on the KeyPad as well
        as the pop-up menus to appear when one of the keys is selected.
   "

   | labels lines selectors keys key |


   labels := #(   'Rectangle\Square\Circle\Ellipse\Polygon\Spline\Line\Lines\Text'
                  'Select\DeSelect\Copy\Move\Delete\Group\UnGroup\Bring  To  Front\Send  To
Back\Color\Fill\Empty'
                  'Save\Load\Merge\New'
                  'Grid\Align'
              ).

   lines :=  #( (4 8) (2 5 7 9) (1 3) () ).

   selectors := #( (addRectangle addSquare addCircle
                    addEllipse addPolygon addSpline addLine addLines addText)

                   (select deselect copy move delete
                   group ungroup bringForward sendBack
                   color fill empty)

                   ( fileOutObjects fileInObjects mergeObjects clear)

                   (grid align)
                 ).

   keys := #( 'Add  Objects...' 'Edit Objects...' 'File...' 'View...' ).

   contents := ''.

   menu := (KeyPad for: self)
      matrixSize: (2 @ 2);
      minimumKeySize:  ListFont charSize + (2 @ 8) * (7 @ 1);
      label: label;
      yourself.
   "
      Add Shape keys.
   "
```

```
    1 to: 4 do: [ :index |
        key :=
            (
             ColoredMenuButton
                labels: (labels at: index) withCrs
                lines: (lines at: index)
                selectors: (selectors at: index)
            )
            foreColor: 6
            backColor: 9;
            id: #yourself;
            contents: (keys at: index).

        menu addKey: key at: ((index - 1 // 2) @ (index - 1 \\ 2))
    ].


    displayPane := ColoredTextPane new
        setForeColor: 0
        backColor: 3;
        model: self;
        name:  #display;
        dispatcher: TextDisplayDispatcher new;
        menu: #noMenu;
        yourself.

    ^menu minimumDisplaySize:
        (TextFont charSize + (2@8) * (25@1));
        notifyApplicationBy: #keySelected:;
        displayPane: displayPane;
        yourself.


keySelected: aKeyID

    "{Dan Peters: Jun 24, 1991 16:00:33}"
    "
          LIMITED PRIVATE
          ===============

          This is the receiver's interface to its KeyPad.  aKeyID
          should be either the name of a Smalltalk class, in which case
          create a new instance of this class, or the name of one of
          the receiver's methods, in which case perform it.  Also,
          bring the graphicsPane to the top of the window stack, and
          update it.
    "
    Scheduler toTop: (graphicsPane dispatcher topDispatcher).

    (Smalltalk includesKey: aKeyID)
    ifTrue: [ self newGraphic: aKeyID]
    ifFalse: [self perform: aKeyID].

    Scheduler dispatchers do: [ :dispatcher |
        dispatcher deactivate
    ].
    Scheduler toTop: (displayPane dispatcher topDispatcher).
    Scheduler topDispatcher activate.
    contents := ''.
    self changed: #display.

>>>>>>>>>>>
```

Sherlock Technical Document

Graphics Editors:
Technical Details

DRAFT

Dan Peters
University Of Pittsburgh
LRDC

July 29, 1991

I. Introduction.

This document is intended to serve as a description of the design and implementation of several very useful graphics tools in Smalltalk V. In the discussion which follows, we will focus our attention on a single application which makes use of these tools, a graphics editor. Our philosophy throughout this design, however, will be to produce tools which are useful for and adaptable to a variety of applications.

In the first section, we will motivate our design with a problem that our tools will be used to solve. This is the graphics editor mentioned above. In this section, we will attempt to identify behavioral boundaries existing within the context of the problem. We will then attempt to generalize these behaviors and use them to isolate candidate classes. From these candidates, we will then select those classes which are already available, those which can be obtained via inheritance, and those which must be implemented from scratch.

Having identified the necessary abstractions, we will then determine the ways in which these items interact. This information will then be used to design a general communication protocol for our classes. This will be one of the more important aspects of the design because it is here that the interfaces to our classes will be specified.

Once this end of the design is complete, we may proceed to a real implementation. In the final section, we will once again turn our attention to our graphics editor application in order to illustrate some of the aspects of the implementation.

II. The Problem.

In this section, we begin the design of several very useful graphics tools. Although we will identify useful tools by examining one particular application, a graphics editor, we will constantly try to provide tools useful in a variety of contexts.

The problem concerning us is to design a graphics editor in Smalltalk V. This application should provide as elementary graphic items, geometric shapes, text, and groups of these items. The user should be able to create these items visually using the mouse. The editor should also provide facilities for moving objects around, copying objects, and removing objects as basic facilities. Additionally, in order to provide a more useful application, there should be available tools which can be used to reshape objects, change the color of objects, place objects in layers file objects in or out, align objects on some kind of coordinate system, and group several objects into a single more complex object.

Let us now begin to sketch out some ideas about implementing this application. Our first concern here is to break the application up into units which have some merit by themselves. The way in which we choose to approach this activity is very important. Since this application lends itself nicely to analysis using object oriented methods, we will attempt to break the application up into useful object type:. We will identify candidate types by noting behavioral boundaries and possibilities for reuse.

The first objects that we will need are centered around displaying the graphics. Here we are in luck, because there are already available several classes which we will find useful for these ends. Specifically, we have **GraphicObject, GraphicElement, GraphicsPane** and its associated Dispatcher, **RegionDispatcher. GraphicElement** is a class used for describing primitive graphics. It can hold information about simple graphics like the location and shape of the graphics, the color of the graphics, and which layer the graphics are to appear in. The **GraphicObject** class is used to collect together one or more graphic elements for display. We can use instances of this class to construct some very complex graphics. A **GraphicsPane** is used to display **GraphicObjects** on the display screen and is useful because it can be used as part of the Smalltalk windowing system as a subpane. Finally, class **Region** is used as a dispatcher for the **GraphicsPane**. There is also the class **GraphicsPen** which knows how to draw **GraphicElements and GraphicObjects** on Forms. These classes are already described in detail in the document Graphics Classes in Smalltalk V by Edward Hughes, so we shall not go into further detail here. The next item that we will need is an interaction mechanism. This mechanism should provide us with a simple method of choosing tools to manipulate our graphics. Here, too, we already have the necessary equipment available. The **KeyPad** interface classes provide us with a rich set of interface possibilities. For the present application, we will choose to use the pull down menu facilities for our purposes. These classes are also described in detail in the document KeyPad: A Versatile Interaction Mechanism, by **Dan Peters**, so we will not pursue this discussion.

By using the facilities described above, we have an almost complete interface available, and we haven't done any real work yet. This illustrates vividly the savings afforded by the object-oriented design paradigm because it strongly supports design for reuse. There are, however, other aspects of the interface which still require specification. First, we would like to be able to change some aspects of the graphics directly on screen. One of the most popular ways of doing this is to denote special points on the object that can be moved around on the screen to change the object in some way. For example, special points on a circle might include its center and several points on its circumference. Using the scheme we are currently considering, we would highlight these points in some way. Then, when the user clicked on one of these points, he could move it around on the screen, and the object would update itself accordingly. For example, if the center was selected and moved, the circle would move so that i^Ts center was now at the new point, without altering its size. If one of the circumference points was chosen, however, the circle's position would remain fixed, but its size would change so that the new point was on the circumference. We will therefore need and object that can provide these facilities for us.

Another interface item that we will require (or would like to have) is some object that can move these graphics around on the screen. In order to be general, this item should also be able to move other displayable items like Forms around on the screen. A similar activity which allows one to define new shapes on the screen is also required. This behavior is already supported by the class, **PointDispatcher**, which allow one to sweep out things like lines, rectangles, ellipses, etc. on the screen.

In addition, since we would like to be able to include text in our graphic displays, we need some mechanism for entering and editing text. We will therefore design a class to perform this function for us.

Once the above items are made available, we will have the interface to our application completed. Beyond that, however, we will have made several useful items available to other applications' interfaces. For example, we will have a generic text-entry\edit facility available. This kind of function is useful in many contexts, so in the future when we need to perform this activity, the facilities will already be on hand. Also, we will have an object which can move displayable objects around on screen. This could be quite useful in many other areas including, for example, making multicolored cursors or cursors which are larger than the current 16x16 limit. There are many other useful applications of the interface devices described above, but let us now return to our design.

When we have the interface classes finished, we will then need some other object or objects available to control the interaction between the items above. In specifying this part of the design, care must be taken to achieve generality. To this end, we elect to break down this control into two distinct areas. The first area will be concerned with object creation and manipulation. The second area will be concerned with organizing and displaying the objects being worked with. For the graphic editor, these activities will be very simple. Objects performing the first task will only need to supply facilities for simple graphic operations. The second task is also very simple in that we will only need some kind of list to hold the objects. By making this second item the model of a **GraphicsPane**, we need worry very little about displaying them. We just hand the list over to the pane and let it do the work.

Since we are designing for generality, however, we can easily envision cases where these two tasks are not so simple. For example, if we wanted to design a graphics-based circuit editor, we would to establish a correspondence between circuit elements and their graphic representations. We would also probably want some sort of router available to route connections, and some sort of simulator so we could check out the circuit function. Finally, we would probably want some facility available which would allow us to easily change component values like impedance. We should therefore keep these things in mind during the design so that building these kinds of applications calls for a minimal amount of excess design. To this end, we will provide hooks on our objects to facilitate communication with things like routers and simulators, and we will specify a generic protocol for using these hooks.

The items discussed above represent the significant portion of the design. Most of what remains (as far as isolating objects) is deciding how to structure our data. We will probably need no new classes for this activity, however.

Let us now proceed to the actual design. In the next section, we will develop a communications protocol for all of our objects. Included in this phase of the design will be the specification of **PUBLIC** methods as well as definition of behavior for each of our classes.

III. Class Interfaces.

Having sketched out a very informal design for our graphics tools, we now proceed to formalize these ideas by specifying the exact behavior exhibited by our classes. We will also describe the **PUBLIC** aspects of our classes. We will begin this discussion by examining each of the objects in isolation. During this discussion, we will not worry about how these objects fit together. After the objects have been thoroughly examined, however, we will show how simply the objects may be pulled together to create the graphics editor application.

A. Class FormMover.

In this section, we consider class **FormMover** in detail. This class will provide us with a mechanism for easily moving displayable items around on the screen according to the cursor's position. Since it interacts with the mouse, an obvious choice for its superclass is the Smalltalk class **Dispatcher**. An instance of class **FormMover** is obtained by sending one of the following messages to the class.

getPointDisplaying: aForm		**move: aGraphicObject**
offset: aPoint	**OR**	**offset: aPoint**
executing: execBlock		**executing: execBlock**
returning: returnBlock		**returning: returnBlock**

The first message above yields an instance of **FormMover** which will display aForm on the screen and cause this display to track the cursor. The image will track the cursor until the mouse is left-clicked. At that time, the returnBlock is evaluated in the context of the object which created the returnBlock. The return block has a single parameter, the current cursor position relative to the position of the form's upper left corner. The returnBlock should end with a forced return like **^self.** This is so that the return is performed in the context of the creator of the returnBlock. If this is not done, the **FormMover** will continue to display aForm.

The parameter **aPoint** allows one to specify the position of the for relative to the cursor. If we want the form's upperleft corner to coincide with the cursor's position, aPoint should be 0@0.

The final parameter, **execBlock**, is a block which we want evaluated each time the cursor's position is updated. It accepts zero, one, or two parameters. The first optional parameter is the current Cursor position. The second is the **FormMover** instance. This last option is of limited usefulness, but it is there in order to correspond with the equivalent block in class **PointDispatcher.**

The second message listed above causes an instance of class **FormMover** to be created also. This instance will draw aGraphicObject on a Form and display this form under the cursor. All other parameters are the same.

Instances of class **FormMover** have no **PUBLIC** messages, since they are createď, do a job, and are then destroyed, never relinquishing control to other objects.

These items are very useful whenever "dragging" an object is desirable. For example, if we were to construct a filing system, we could use a **FormMover** to implement the moving around of folders and files in a graphical way. The objects could also be used to create much more complex cursors, since they support multiColored Forms, as well as Forms of arbitrary size.

B. Class ActiveObject.

Instances of class **ActiveObject** perform the function of allowing one to graphically alter the characteristics of objects. An instance of **ActiveObject** is obtained by sending the message

on: anObject
for: owner
notifyBySending: aMessage

to the class. **anObject** should be the object which is to be changed. This object should answer the message **graphicObject** with an instance of class **GraphicObject**. This graphic object will be queried for its set of **change points**, and these will be used to effect changes.

The **owner** parameter indicates the object that is to be informed in case any changes are detected. When a change is detected, the **owner** will be sent the message named by **aMessage**. **aMessage** should accept two parameters, aPoint and aShape. The point will be the new location of the changed point, and aShape will be the geometric shape to which the changed point belongs. At this time, the **ActiveObject** will also send the message **replace: oldPoint with: newPoint** to the shape indicated. All geometric currently implement this method.

The points available for changing are highlighted with a small square over the point. When the user clicks inside the area covered by the square, the square tracks the cursor until the next click. At this click, the owner will be sent the message **aMessage** with the indicated parameters.

The only **PUBLIC** method available on instances of class **ActiveObject** is the **update** method. Sending this message to the **ActiveObject** will cause all of the highlighted points to be reacquired and updated.

C. Class TextPad.

The **TextPad** class provides a text interaction tool for applications. An instance is obtained by sending the message

in: aBox **for: anObject**	OR	**in: aBox** **editing: aString**

```
saveWith: saveSymbol        for: anObject
abortWith: abortSymbol              saveWith: saveSymbol
                            abortWith: abortSymbol
```

The first method yields a **TextPad** with no contents. The instance will be bounded by **aBox**. When the **TextPad** is opened, the user can enter and edit text in the familiar way. There are, however, two "buttons" at the bottom of the **TextPad**. The first, **SAVE**, causes the message named by **saveSymbol** to be sent to **anObject.** This message should accept three parameters, a string holding the contents of the **TextPad**, the **TextPad** itself, and the **TextPad's** bounding box. After this is done, the **TextPad** will close itself. The other button is labelled **CANCEL.** Pressing this button will cause the **TextPad** to close itself without reporting to anObject.

The second method above will do essentially the same thing, except the original contents of the **TextPad** are the characters in aString.

These items can be used anytime text needs to be entered independently of some main program. **anObject** simply creates one of these items and promptly forgets about it and goes on to other things. When the user clicks **SAVE** the **TextPad** will notify **anObject** via the indicated message. **anObject** does not need to know anything else about the process.

D. Class MenuDriver.

We have now discussed several of the smaller utility classes useful for our purposes. We will now move to a higher level and discuss the controlling object class. It is here that the resources of the above object are exploited and coordinated.

Objects of class **MenuDriver** are primarily useful in coordinating the activities of other utility objects. The class itself is abstract, but at present it has two subclasses, **GraphicsBuilder, and DeviceBuilder**. The subclasses embody knowledge specific to the objects that they are designed to create. The superclass, however, provides a set of behaviors common to many different types of editors.

An instance of **MenuDriver** can be obtained by sending the following message to the **MenuDriver** class:

```
for: anObject
on: aPane
labelled: aString.
```

anObject will be some object that knows about organizing and displaying the items which **MenuDriver** is responsible for manipulating. **aPane** is the pane in which the **MenuDriver**'s items are being displayed. This is so that the **MenuDriver** can update the display independently of other objects. **aString** is a label which will appear atop the **MenuDriver**'s menu.

After creating the **MenuDriver**, several other pieces of information are required before it can do its work. This information is defined by the application. Since this is very important, we will spend some time describing this in detail.

MenuDrivers operate according to an **event driven** schedule. This means that these objects will do nothing until an **event** occurs. Now an **event** is some activity that is important to the application. For the graphics editor, some important **events** might be the addition, deletion, or changing of some graphic object. In fact, the subclass **GraphicsBuilder** defines such events and names them

#GraphicAdded, #GraphicDeleted, and #GraphicChanged respectively. When a **GraphicsBuilder** is created, the creating object notifies it of which of these events it wants to be made aware of. It does this by sending the message **when: anEvent perform: aMethod** to the **GraphicsBuilder.** Now whenever the **GraphicsBuilder** detects the occurrence of **anEvent**, it will send the message **aMethod** to its owner. When subclassing **MenuDriver**, one of the most important details to consider is which events to handle, and how to inform the owner of their occurrence. For example, in the graphics editor when an object is to be deleted, the **GraphicsBuilder** will check to see if its owner wants to be notified of this event. If not, no further action is taken. If the event is important, however, the owner will be sent the required message. Now **GraphicsBuilders** assume that the message accepts a single parameter, the object which is to be deleted. It is up to the owner to actually perform the deletion. This is because the **GraphicsBuilder** assumes no responsibility for the organization or display of the objects. It is assumed that the owner manages this completely.

This technique of responding to important events is very powerful, allowing applications the freedom to define important events and respond to them in any way the see fit. In addition, very little overhead is introduced in creating custom **MenuDrivers** because the event detection and function dispatch are already provided. They are also parameterized to allow for extreme flexibility. Therefore, one need not concern oneself wit these details. Rather, a designer can focus on the task before him and spend very little time on communication details.

This technique also decouples the object manipulating the items and the object organizing them. We can therefore attach one of these **MenuDrivers** to several types of organizers like simple editors, databases, or even other applications like image processing packages.

We have now discussed all of the classes that must be implemented via inheritance. It would, however, be very useful to have one further class that could edit portions of objects which are not easily represented by graphics. Such a class is available but will not be discussed at length here. Instead, let us turn to a discussion about how all of these items tie together.

E. The Graphics Editor.

In this section, we will discuss the means by which all of these items can be made to work together. We will use the graphics editor as a simple example. The techniques are generally applicable, however.

The Graphics Editor represents a unification of many classes mentioned above as well as several which were only hinted at. The main classes used to build the editor are **GraphicsEditor, and GraphicsBuilder.** The **GraphicsEditor** class is used as the container\display manager mentioned above. One of the options available to users from its pane menu is **Open Graphics Menu.** Selecting this option will create a new instance of **GraphicsBuilder.** The **GraphicsBuilder** is told to operate **for:** the **GraphicsEditor** and with respect to this object's graphics pane. **The GraphicsEditor** further signifies the events **#GraphicAdded, #GraphicDeleted, and #GraphicChanged** as important by using the **when:perform:** protocol. After this, control is transferred to the **GraphicsBuilder** by instructing it to open its menu. Now a menu is opened. This menu is an instance of class **KeyPad** and has as keys four **MenuButtons.** These buttons group together the various activities supported by the **GraphicsBuilder** into the groups **Add Graphics, Edit Graphics, File, and View.** Selection of any item under **Add Graphics** is interpreted as the **#GraphicAdded** event. Most of the items under the

Edit Graphics menu are interpreted as the **#GraphicChanged** event. The **#GraphicDeleted** event corresponds to selection of the **delete** item under **Edit Graphics**. Some of these items, although primarily of one type of event, also cause other events to occur. For example, choosing **group objects** is interpreted as changing one object and deleting several others (perhaps). We can see therefore how the events are detected (by menu choice) and responded to (by choice interpretation). When subclassing **MenuDriver**, this will be the most important part of the design.

How are the other classes considered above used? These are important in implementing the **PRIVATE** methods on **GraphicsBuilder**. These items are used by the **GraphicsBuilder** to perform services which are a part of its response to events. This delegation of responsibility frees the **GraphicsBuilder** from having knowledge about many different activities like getting text or moving forms, so that it can focus primarily on event detection and response. It would also make very little sense to include any of these other tasks under **GraphicsBuilder** since that would render them useless in other contexts. In other words, if **GraphicsBuilder** knew about moving forms, any other application that needed to do this would have to either reimplement the behavior already available in **GraphicsBuilder**, or use a **GraphicsBuilder** (in a manner for which it was not designed) to perform the required task. This would clearly be poor design since it minimizes reuse and clouds the function of the **GraphicsBuilder** object.